Ada in Action

(with Practical Programming Examples)

by

Do-While Jones

Second Edition

1995

Formatting by Herman Claus

Printing Version 1 - created : 4 April, 1996

CONTENTS

Figures

Listings

Copyright Notice (Do-While Jones)

The book, Ada in Action (with Practical Programming Examples) is filled with copyright notices that say:

```
--  Copyright 1989 by John Wiley & Sons, Inc.
--         All Rights Reserved.
```

However, there was a clause in my publication contract that said that if John Wiley & Sons let the book go out of print, I could request them to reprint it. If they failed to reprint it in 1 year, the copyrights would be assigned back to me.

Ada in Action went out of print. I requested a reprint. John Wiley & Sons refused to reprint it, and sent me a letter (which I have on file) transferring the copyrights back to me.

The new copyright notice for the book and source code listings is:

Copyright Notice & Remarks (Herman Claus)

All information mentioned on this page was valid on 1996-4-1.

Original ASCII text version of the book 'Ada in Action' can be found at :

ftp:\\ridgecrest.ca.us/pub/users/d/do_while

An HTML version can be found at http://www.cs.wm.edu/~collins/ada/ada-book/book.html. You cannot download the whole set of files, so you're forced to stay on-line. Check also ada belgium at http://www.cs.kuleuven.ac.be/~dirk/ada-belgium/aia.html

I downloaded the ASCII version of the book, formatted it and put it in a PDF format because I don't like reading this kind of texts on a screen, and I surely don't want to stay connected to Internet while browsing the HTML version. I hope you enjoy it.

If you have any comments or corrections, please send an E-mail to :

(1) do-while jones at do_while@ridgecrest.ca.us

(2) Herman Claus at P82798@vnet.atea.be

Mention ada_in_action as a subject.

I used the hard- and software on the job for creating this, but the time was of my own. As a compensation for the company, a small commercial. I'm working for Siemens Atea n.v., a belgian telecom company, fully owned by Siemens. My division produces a 1.2 Gbps optical network (called OTN) that can transport voice, video and data over a few thousands of kilometers, and that is redundant and self healing too. We do high quality digital video and digital audio, 2- and 4-wire telephone and ISDN, Token Ring, Ethernet and RS232/RS485. We're rather proud of it. We mainly sell to subways, pipelines, large infrastructures. Check the WWW -> we should be there anytime now.

Dedication for the First Edition

This book is gratefully dedicated to God, in the hopes that it will be used

for attaining wisdom and discipline;
for understanding words of insight;
for acquiring a disciplined and prudent life,
doing what is right and just and fair;
for giving prudence to the simple,
knowledge and discretion to the young--
let the wise listen and add to their learning,
and let the discerning get guidance--
for understanding proverbs and parables,
the sayings and riddles of the wise.

(Proverbs 1:2-6 New International Version)

Dedication for the Second Edition

The second edition is lovingly dedicated to my wife.

A wife of noble character who can find?
She is worth far more than rubies.
Her husband has full confidence in her
and lacks nothing of value.
She brings him good, not harm,
all the days of her life.
She selects wool and flax
and works with eager hands.
She is like the merchant ships,
bringing her food from afar.
She gets up while it is still dark;
she provides food for her family
and portions for her servant girls.
She considers a field and buys it;
out of her earnings she plants a vineyard.
She sets about her work vigorously;
her arms are strong for her tasks.
She sees that her trading is profitable,
and her lamp does not go out at night.
In her hand she holds the distaff
and grasps the spindle with her fingers.
She opens her arms to the poor
and extends her hands to the needy.
When it snows, she has no fear for her household;
for all of them are clothed in scarlet.
She makes coverings for her bed;
she is clothed in fine linen and purple.
Her husband is respected at the city gate,
where he takes his seat among the elders of the land.
She makes linen garments and sells them,
and supplies the merchants with sashes.
She is clothed with strength and dignity;
she can laugh at the days to come.
She speaks with wisdom,
and faithful instruction is on her tongue.
She watches over the affairs of her household
and does not eat the bread of idleness.
Her children arise and call her blessed;
her husband also, and he praises her:
"Many women do noble things,
but you surpass them all."
Charm is deceptive, and beauty is fleeting;
but a woman who fears the Lord is to be praised.
Give her the reward she has earned,
and let her works bring her praise at the city gate.

(Proverbs 31:10-31 New International Version)

Chapter 1. INTRODUCTION

This book is not intended to teach you the Ada programming language. You should already be familiar with Ada syntax and semantics. My goal is to share with you the experiences I've had using Ada in engineering applications. I hope these pages will help you avoid some common pitfalls. Most of all, I hope I can help you fill your bag-of-tricks with some reusable Ada routines.

1.1. Organization and Content

The rest of this book is divided into four main topics. The first topic is numeric considerations. The examples Chapter 2 illustrate the things you need to think about whenever your program does non-trivial calculations. This includes obvious things like how many bits you will need for integers, and what floating-point data type to use, but it also includes some things you probably haven't been exposed to before. The idea of letting the compiler check the consistency of the dimensional quantities in equations is a new innovation made possible by the Ada language.

Another difficult problem most programs have to deal with is the user interface, so it is the second topic. People aren't as predictable and consistent as mechanical devices are, which makes user interfaces difficult to design. This is an area with a lot of potential for reusable software. Chapter 3 is full of utility routines that I think you will find very useful.

Contrary to what you've often heard, the whole is more than the sum of its parts. Even if you have all the pieces, they aren't worth much if you don't know how to put them together. Chapter 4 shows several examples of small-scale programming, and one example of more rigorous software engineering, which is the third topic.

The last topic is testing. I saved it for last because writing code is easy; making sure it works correctly is hard. Over the years I've used a variety of methods to check code, and Chapter 5 talks about them.

A quick peek at the back of the book shows there are hundreds of pages of examples of Ada source code. The first examples are simple, and they get more complex as you read toward the end of the book. "Complex" has come to mean "difficult", but that's not the case here. The later examples are properly called complex because they were composed by combining the smaller building blocks found near the beginning to create bigger, more powerful, building blocks. This means it will be easiest to understand if you read the book from front to back without skipping around.

When choosing the examples, I didn't go through the list of Ada reserved words and try to come up with an example for each one. If I did, the first example would have an abort statement in it, the second would show how to use the abs operator, and so on. These contrived examples wouldn't do much more than show syntax, which you should already know.

Contrived examples often do things strange ways just for the sake of illustrating a point, and this sometimes teaches bad programming habits. For example, recursion is often demonstrated by recursively computing N factorial (N! = N * (N-1)! until N = 1). That's a good way to show recursion, but a terrible way to compute N factorial. Students often miss the point of the example. If the homework assignment is to write an excellent program to compute N factorial, guess how most of the students will do it. (Hint--They don't use a loop!)

I used practical examples with real applications, so the examples determined what Ada features were demonstrated and their order of appearance. This approach gives the most exposure to the most commonly used Ada features, and less emphasis on the less important ones. The only two Ada features that didn't get a fair shake from this approach were tasking and access types.

There are so many things that need to be considered when using tasking, I might be able to devote almost an entire book to it. I wanted to avoid any discussion of tasking because I didn't want to get started on a topic I didn't have space to adequately explain. (It turned out that I couldn't avoid tasking completely, so you will find a brief example in the VMS package.)

Access types are most useful for solving problems in very specialized areas. The only time I've ever needed access types was for an Artificial Intelligence (AI) problem. My AI example is an excellent showcase of access types, but it requires too much technical AI background to fit in this book, and it isn't quite ready for publication yet. That's why you won't find any examples of access types in this book.

1.2. Figures and Listings

All the figures and listings are at the back of the book for easy reference[1]. Despite their location, they are meant to be read right along with the text. Package specifications include comments that tell what the routines do, and how to use them. Comments in the package bodies tell how the algorithms work. I don't generally repeat this information in the body of the book. The body of the book tells what the other alternatives were and why I made the decisions I did. If you don't read the figures and listings when they are first mentioned in the text, you might not understand the discussion.

There is a distinction between figures and listings. Both contain code, but the code in the figures are not intended to be used in your application programs. Some of the figures are examples of what not to do. Other figures are examples of correct style, but are intentionally incomplete (to avoid obscuring the main point of the figure with necessary but extraneous details). Listings, however, are complete. They have been compiled and tested on at least one validated Ada compiler, and may be useful in your application programs. The source code for all the listings (but not the figures) can be purchased separately on a floppy disk.

1.3. Copyright

It would be foolish for me to write a book revealing many of my software secrets, and then say to you, "You may buy this book only if you promise never to use any of the code I'm going to show you." It would be even more foolish of you to buy the book under those conditions. I expect (and want) you to use the code shown in the listings in your programs. If you make a bundle of money doing that, good for you. You don't owe me any royalties.

1.4. Liability

As far as I know, there are no errors in the listings, but I have no control over them once they leave my computer. There may be a typesetting error, or you may miscopy the listings. The source code disk may be defective, or your disk drive may introduce undetected errors when reading it. You may modify the listings in some way that introduces an unexpected side effect. You could compile them with a defective compiler. There may be a strange run-time dependency in your computer that my computer doesn't have. There are all sorts of things that could go wrong, over which I have no control.

Since I have no way of knowing how you are going to use the code, what compiler you are going to use to compile it, what target machine it will run on, or how you are going to modify it, I can't guarantee that it

[1]Not true anymore : I placed them in the text - this seemed easier to me for reading (H. Claus)

will work in your application. You may use my code at your own risk but you have to take the responsibility for the consequences yourself.

1.5. Ada not ADA

Everyone reading this book should know that ADA is the American Dental Association and Ada is a programming language named in honor of Ada Lovelace, the first programmer. Ada was born in 1815, and that's the reason the Ada Language Reference Manual (LRM) is MIL-STD-1815A. In light of this distinctly feminine background of the language, I consistently use feminine pronouns when referring to Ada. To avoid confusion, I use masculine pronouns for humans (programmers and users) of either gender.

Chapter 2. NUMERIC CONSIDERATIONS

Granted there are some programs that don't have to do extensive mathematical calculations, but as a general rule computers have to manipulate numbers to get results. This section is devoted to things you have to consider when working with numbers.

2.1. POOR_COORDINATES package

Embedded computers often have to convert from rectangular coordinates to polar coordinates to track a target or compute an intercept trajectory. Real applications generally use three dimensional geometry, but to keep the example as simple as possible I have used only two dimensions. The POOR_COORDINATES package shown in Figure 1 is typical of how a beginning Ada programmer would write the code. It isn't a terrible package, but it could be a lot better.

There are several things right in Figure 1. Credit must be given for recognizing that transformations between coordinate systems are common to a variety of embedded computer applications, and putting them together in a library package makes them available for reuse by future programs. It was perceptive to recognize the need for data types representing rectangular and polar data points, and those type definitions certainly belong in the POOR_COORDINATES package. It is highly commendable that comments were used to tell us that the distances are in feet and the angles are in degrees. The package specification is in its own file, so it can be compiled separately. These are all good software engineering practices that are usually taught in introductory Ada classes.

2.1.1. Control of Integer Data Length

Even so, there is a fatal flaw in the POOR_COORDINATES package specification. There is likely to be trouble if this package is developed on a host computer and then recompiled for a different target. Suppose the software is developed on a VAX. Since the VAX is a 32 bit machine, VAX Ada compilers naturally use 32 bits for the predefined type integer, The distance (in feet) that can be represented by 32 bits is nearly a round trip from the earth to the moon.

The predefined integer type doesn't always have to be 32 bits. The Meridian and Alsys Ada compilers for 8086 machines use 16 bits for integer. The distance (in feet) that can be represented by 16 bits is just over

Figure 1. POOR_COORDINATES package specification.

```
------------------------------------------------------------

package POOR_COORDINATES is

  type Rectangular_points is
    record
      NORTH : integer; -- feet;
      EAST  : integer; -- feet;
    end record;

  type Polar_points is
    record
      R     : integer; -- feet;
      THETA : float;   -- degrees;
    end record;

  function Transform(RP : Rectangular_points)
    return Polar_points;

  function Transform(PP : Polar_points)
    return Rectangular_points;

end POOR_COORDINATES;
```

```
Figure 2. Distinct types.
------------------------------------------------------------
-- Each declared type is distinct, even if they have the
-- same name and definition as another integer type and are
-- in the same file.

-- In the example below, the two packages X and Y, and the
-- procedure P, are in a file called XYP.ada.

--             XYP.ada

package X is

  type Whole_numbers is range -32768..32767;

  procedure Fool_With(N : in out Whole_numbers);

end X;

package Y is

  type Whole_numbers is range -32768..32767;

  procedure Produce(Z : out Whole_numbers);

end Y;

with X, Y; use X, Y;
procedure P is

  type Whole_numbers is range -32768..32767;

  W : Whole_numbers;

begin
  Produce(W);   -- this is line 28
  Fool_With(W); -- this is line 29
end P;

----------------------------------------------------
When you compile the file XYP.ada above, here's what
you get:

Meridian AdaVantage(tm) Compiler [v1.5 Apr  3, 1987] Target
8086
Package x added to library.
Package y added to library.
"XYP.ada", line 28: <<error>> identifier has wrong type "w"
"XYP.ada", line 29: <<error>> identifier has wrong type "w"
31 lines compiled.
2 errors detected.
```

6 miles. That often isn't enough. Programs that use the POOR_COORDINATES package may work on a VAX, but not an 8086 because the number of bits in the predefined integer type is machine dependent.

The Ada Language Reference Manual allows compilers to predefine families of integer data types with names like integer, long_integer, long_long_integer, short_integer, and short_short_integer, but it doesn't say how many bits to use. A long_integer on an IBM PC has the same number of bits as an integer on a VAX.

2.2. STANDARD_INTEGERS package

Whenever it matters how many bits there will be in an integer, you should always specify the range yourself. If you try to do this locally in your program, then you might have some trouble interfacing with a

Figure 3. SLIGHTLY_BETTER_COORDINATES package specification.

```
----------------------------------------------------------
-- This package is better because it will always use
-- integers with 32-bit range, regardless of the computer.

with STANDARD_INTEGERS; use STANDARD_INTEGERS;
package SLIGHTLY_BETTER_COORDINATES is

  type Rectangular_points is
    record
      NORTH : Integer_32; -- feet;
      EAST  : Integer_32; -- feet;
    end record;

  type Polar_points is
    record
      R     : Integer_32; -- feet;
      THETA : float;   -- degrees;
    end record;

  function Transform(RP : Rectangular_points)
    return Polar_points;

  function Transform(PP : Polar_points)
    return Rectangular_points;

end SLIGHTLY_BETTER_COORDINATES;
```

library unit. Consider Figure 2, where package X has declared a 16-bit integer, and package Y has declared a 16-bit integer, and procedure P wants to use both packages at once. The types X.Whole_numbers and Y.Whole_numbers aren't the same type as Whole_numbers, so an Ada compiler will give an error similar to the one shown in Figure 2.

The problem can be solved by using the package STANDARD_INTEGERS, given in Listing 1. It declares some useful integer types of known ranges which can be shared by everybody. I didn't use a representation clause to specify the number of bits because I don't really care how many bits the computer uses. If a 32-bit computer finds it easier to use a whole 32-bit word to store an 8-bit value, that's fine with me. The important thing is to set the allowable range of values. Whether the computer has to use partial words or multiple words isn't of any concern unless it causes a performance limitation. (If I expect to run short of space or time, then I will make representation suggestions to the compiler, but under normal circumstances that isn't necessary.)

The SLIGHTLY_BETTER_COORDINATES package in Figure 3 shows how to use STANDARD_INTEGERS to solve the machine specific range problem.

2.2.1. Shared Data Types

Since Integer_16 is declared only once, any package or procedure that is compiled in the context of STANDARD_INTEGERS can use them. Figure 4 shows how packages X and Y and procedure P can share STANDARD_INTEGERS.Integer_16.

Engineering application programs often have one package that defines some data types that will be shared by many units. That's a consequence of the effect we saw in Figure 2, and it is an intentional feature of the Ada language. It's good to force the definition of a type to be in one place rather than allowing duplicate definitions to exist in several places. There's a chance that some, but not all, of the duplicate definitions would get changed when fixing a bug (creating even more bugs). The single definition of a shared data type insures that all modules will be working with the same kind of data. Putting all the type definitions in a single package makes them easy to find.

```
Figure 4. Shared types.
------------------------------------------------------------
-- The units X, Y, and P in file XYP2.ada can all declare
-- objects of type Integer_16 because they can share the
-- declaration of that type in STANDARD_INTEGERS.

--             XYP2.ada

with STANDARD_INTEGERS; use STANDARD_INTEGERS;
package X is

  procedure Fool_With(N : in out Integer_16);

end X;

with STANDARD_INTEGERS; use STANDARD_INTEGERS;
package Y is

  procedure Produce(Z : out Integer_16);

end Y;

with X, Y, STANDARD_INTEGERS; use X, Y, STANDARD_INTEGERS;
procedure P is

  W : Integer_16;

begin
  Produce(W);
  Fool_With(W);
end P;
```

There is a danger, however, in creating one monster type definition package. Monsters usually try to take over the world, and type definition packages are no exception. If you put all your type definitions in one package, sooner or later practically every module depends on it. Then every module must be recompiled whenever a new type is added to the monster.

It is usually better to have several smaller type definition packages, and let the application program use whatever packages it needs. For example, you might have a package BRITISH_UNITS with data types Feet, Pounds, Seconds, and so on, that you have used on several projects. If you are assigned a new project that needs metric types as well, don't add the new metric types to BRITISH_UNITS. If you do, you will have to recompile everything that uses BRITISH_UNITS. Instead, write a new package METRIC_UNITS with types Meters and Newtons in it. Don't put Seconds in METRIC_UNITS because it already exists in BRITISH_UNITS, and Ada will try to keep BRITISH_UNITS.Seconds distinct from METRIC_UNITS.Seconds. If your program needs Newtons, Meters, and Seconds, compile it in the context of both BRITISH_UNITS and METRIC_UNITS.

Some data types are more naturally defined in a special package, rather than a general data types package. For example, it makes more sense to define Rectangular_points and Polar_points in POOR_COORDINATES (as was done in Figure 1) than it does to put them in BRITISH_UNITS and then compile POOR_COORDINATES in the context of BRITISH_UNITS. If POOR_COORDINATES was modified to use floating point types instead of Integer_32, then you would have to recompile BRITISH_UNITS and dozens (maybe hundreds) of modules that depend on BRITISH_UNITS. Leaving Rectangular_points and Polar_points in POOR_COORDINATES, allows you to change their definition without affecting many unrelated modules. You only need to recompile the few modules that depend on POOR_COORDINATES. That's a better approach to take.

2.3. Non-existent STANDARD_FLOATS package

Since I use a package called STANDARD_INTEGERS, it is logical for you to assume I have also written a package called STANDARD_FLOATS with data types Float_7 and Float_15 defining seven- and fifteen-digit real numbers. I haven't and don't intend to, because I don't think it is a good idea. To explain why, I need to tell you a little personal history.

I went to school long ago, in the years B.C. (Before Calculators.) In those days you could easily spot an engineering student by the slide rule dangling from his belt like a sword. The slide rule was his weapon for slaying the dragons he encountered daily. It was a mechanical device with numbers and lines etched on it, which was used for multiplying, dividing, and computing trig functions. It gave three digit accuracy, and you had to keep track of the exponent in your head.

Three digits let you express numbers from 00.0 to 99.9, so the best accuracy you could get from a slide rule was 0.1%. That never bothered civil engineers when they calculated the maximum load a bridge could stand, because they knew better than to design the bridge with only a 0.1% safety factor. The limited accuracy of calculations forced engineers to be conservative, and use a little bit of common sense.

I fear that younger engineers, lacking that heritage, have gone digit crazy. A fourteen-digit calculator has a subtle way of seducing people into unrealistic conclusions.

2.3.1. Type float Default Precision

Ada's predefined type float can be any number of digits selected by the implementer. Generally hardware considerations or software compatibility issues limit the available choices. For example, VAX/VMS has standard floating-point representations which are 6, 9, 15, and 33 digits of precision, so Ada maps all user-defined floating- point types onto one of those representations.

I have not yet found an implementation that uses fewer than 6 digits for the predefined type float. I maintain that 6 digits is plenty of resolution, and I am skeptical of designs that claim to require more.

Suppose we want to represent angles in degrees using floating-point numbers. If we use 6 digits of resolution, we need 3 digits to represent whole degrees up to 360, leaving 3 digits for fractional degrees. Angles can be resolved to 0.001 degree. Will this precision yield good enough accuracy? In real applications it better.

Let's say that you are using these angles in some avionics software that slaves the infrared seeker on a missile to a position determined by the aircraft radar so the missile can lock onto the target. If this system requires 0.001 degree accuracy to work, it means that the radar and missile pylon have to be aligned to the aircraft body to within 0.001 degree. Do you think a sailor on an aircraft-carrier (who has been on combat alert and hasn't had much sleep lately) can do that on a rolling aircraft carrier deck in a few minutes? Even if he can, will it still be aligned after a catapult launch and an arrested landing? Don't kid yourself. If you design a system that can't tolerate 0.001 degree error, it is never going to work in combat.

Can anyone argue that it is ever necessary to compute the sine of an angle to 15 decimal places? Sines are used as scale factors. For example, DISTANCE_NORTH := RADIAL_DISTANCE * Cos(BEARING);. Does the scale factor really need to be accurate to 0.000_000_000_000_1 %? Was RADIAL_DISTANCE measured to that accuracy? Care to figure out how accurately you have to measure the BEARING to maintain that accuracy?

The reason I don't have a STANDARD_FLOATS package is that I don't believe in designing programs that need floating-point numbers with better resolution than 1 part per million. There may be applications in research laboratories that need higher precision, but in my applications the input data is only good to "slide rule accuracy." I use the default type float because the compiler has probably selected the representation that gives the fastest computational speed.

2.4. DIM_INT_32 Package

STANDARD_INTEGERS solves machine-specific range problems, but there is another dragon to slay before we are completely safe. We need to protect our program against dimensional- unit errors.

Figure 5 uses a package called DIM_INT_32. It is the skillful blend of derived types, private types, and a generic unit. We are about to embark on the derivation of that package. I've let you peek ahead because I wanted to give you a glimpse of how easy and valuable it is to use dimensional units. The derivation of this package is not trivial, but once derived it is simple to use. All you need is one context clause and one type definition for each kind of unit you want to use. The benefit you reap is the detection of dimensional inconsistencies at compile time.

```
Figure 5. Dimensional units example.
------------------------------------------------------------
-- Ada can spot equations that are dimensionally incorrect
-- and add correct conversion factors at compile time.

--             D_U_EX.ada

with STANDARD_INTEGERS, DIM_INT_32;
procedure Dimensional_Units_Example is

  type Feet is new DIM_INT_32.Units;
  type Meters is new DIM_INT_32.Units;

  A, B, C : Feet;
  X, Y, Z : Meters;

  function Units_Convert(M : Meters) return Feet is
    use STANDARD_Integers; -- for multiply
    DISTANCE : Integer_32;
  begin
    -- 1 meter is approximately 3 feet.
    DISTANCE := 3 * Dimensionless(M);
    return Type_Convert(DISTANCE);
  end Units_Convert;

begin
  A := B + C;
  X := Y + Z;
  A := B + Y; -- this line (25) is wrong
  A := B + Units_Convert(Y);
end Dimensional_Units_Example;

------------------------------------------
The above code is in a file called D_U_EX.ada. When you
compile it, here's what you get:

C:>ada D_U_EX.ada
Meridian AdaVantage(tm) Compiler
[v2.1 Feb 29, 1988] Target 8086

"D_U_EX.ada", 25: type of function
does not match context "+" [LRM 6.4]

28 lines compiled.
1 error detected.
```

2.4.1. Type Checking

Have you spent many hours debugging a program before you discovered the angular argument to a trig function was expressed in degrees instead of radians? Has a misplaced parenthesis ever caused one of your equations to produce strange results? Most of these kinds of errors can be detected at compile time if dimensional data types are used.

For example, consider the equation DISTANCE := (INDICATED_AIR_SPEED + WIND_SPEED) * TIME;. This equation will be correct only if the variables use consistent units (i.e. DISTANCE in feet, the two speeds in feet/second, and TIME in seconds). If TIME is in milliseconds, INDICATED_AIR_SPEED is in knots, and WIND_SPEED is in miles per hour, then DISTANCE can't possibly be correct. Even if consistent units are used, you won't get the correct answer if the parentheses are missing because the multiplication operation has higher precedence than addition.

If the compiler knows that two variables of type feet per second added together produce a result of type feet per second, and if it also knows that a variable of type feet per second multiplied by a variable of type seconds produces a result in feet, then the compiler can check the dimensional consistency of an equation for you. If you leave out the parentheses in the example above, the compiler will know that WIND_SPEED multiplied by TIME gives an answer in feet, and feet added to INDICATED_AIR_SPEED in feet per second won't give a correct answer in feet.

Ada's built-in type checking enables her to automatically check the dimensional consistency of all your equations at compile time, if you simply declare the variables to be dimensional units.

Let's look at Figure 5 closely to see how this is done. It uses a package called DIM_INT_32, which contains a dimensioned 32 bit data type Units. Feet and Meters are both new types derived from Units. The variables A, B, and C express distances in feet, but X, Y, and Z express distances in meters. Lines 13 and 14 of Figure 5 show that Ada's type checking will allow addition of distances in the same kind of units, but will prohibit you from accidentally adding distances in mixed units. If you really want to add a distance measured in feet to a distance in measured in meters, you can use Units_Convert to tell Ada you really want to do that, and she will automatically multiply by the proper scale factor.

2.4.2. Derived Types

A derived type has all the allowable values and operations of its parent type, but is an entirely different type. You can derive a type from any other type. You may wonder why I derived the type Feet from the type Units. Why not just derive it from integer?

Suppose I wrote type Kilograms is new integer;. This would create a new data type which could have any value from integer'FIRST to integer'LAST. It would also have all the integer operations defined for it. You could add two objects of type Kilograms and get a result of type Kilograms. You could not, however, add an object of type Kilograms to any other type object, integer or otherwise. This gives us some of the protection we desire.

Deriving Kilograms from integer does not give us a complete solution. Kilograms could be multiplied by Kilograms to get an incorrect answer in Kilograms because integers can be multiplied together, and so that property is automatically derived from the parent. Although Ada always allows you to add new operations to a data type, she never lets you take away operations derived from a parent. We can't derive dimensional units from the predefined integer data type and then remove unwanted operations. We have to start with a data type with fewer operations than we need, and add more operations to it.

2.4.3. Private Types

Private types have no predefined operations except assignment, equality, and inequality. Those are useful operations for dimensional data types, so we are glad to inherit them. But private types don't have an addition operation defined for them. Therefore we have to specially define addition.

I published package called DIMENSIONAL_UNITS in the spring of 1987 ([2]) and put it in the public domain Ada Software Repository ([3]). This package used private types as the parent types for integer and floating point dimensional units. It bothered me, however, that I lost automatic range checking for dimensional data types. That is, I couldn't define a data type Kelvin that automatically detected impossible (negative) temperatures.

I also didn't like the excessive size of the DIMENSIONAL_UNITS package. I put both integer and floating point data types in the package, so programs that needed only integer types also picked up some dead code associated with floating point types.

The DIMENSIONAL_UNITS package works fine, but there was room for improvement. The two generic packages (and their instantiations) that you are about to see include those improvements. They allow you to add range constraints, and separate the integer data types from the floating point data types, so you can selectively obtain the data types you need.

The Range_Checking_Example in Figure 6 shows how Ada can detect impossible values. In the declarative region I instantiated the generic FLOAT_UNITS package to get a new package called TEMPERATURE. I put range constraints on it by specifying MIN and MAX values. The MIN value is absolute zero on the Celsius scale. The MAX value is arbitrary. I could have used the default value, but you wouldn't have known what that value that was. I picked 32_000.0 for no particular reason.

In the body of the code I have three assignment statements, each enclosed in a block structure with an exception handler. You can consider them to be if-then-else structures. If the assignment statement succeeds, then the following line is printed. If the attempted assignment raises CONSTRAINT_ERROR, then the program jumps to the exception handler and prints the line found there.

The lines of output appended to the end of Figure 6 shows Ada found the errors when I ran the program.

[2] Do-While Jones, "Dimensional Data Types," Dr. Dobb's Journal of Software Tools, #127 May 1987, pp.50-62

[3] The Ada Software Repository is a collection of files containing reusable Ada components. These files are maintained by Richard Conn on a computer named SIMTEL20. SIMTEL20 can be reached by anyone who has a computer on the ARPA/MILNET network and File Transfer Protocol (FTP). See Richard Conn's The Ada Software Repository and the Defense Data Network, published by New York Zoetrope, 838 Broadway, New York, NY 10002.

```
Figure 6. Range checking example.
------------------------------------------------------
-- Ada knows that objects of type Celsius can
-- never be colder than absolute zero, and will
-- raise CONSTRAINT_ERROR if an attempt is made
-- to assign a value that is out of range.

with FLOAT_UNITS;
with TEXT_IO; use TEXT_IO;
procedure Range_Checking_Example is

  package TEMPERATURE is new FLOAT_UNITS
    (Float_type => float,
     MIN => -273.16,
     MAX => 32000.0,
     Integer_type => integer);
  type Celsius is new TEMPERATURE.Units;
  LAB_TEMPERATURE : Celsius;

begin
  begin
    LAB_TEMPERATURE := Type_Convert(25.0);
      put_line("Ada will let you assign"
            & " reasonable values.");
  exception
    when others =>
      put_line("Ada FAILED to assign a correct value.");
  end;
  begin
    LAB_TEMPERATURE := Type_Convert(-300.0);
    put_line("FAILED to detect too cold.");
  exception
    when CONSTRAINT_ERROR =>
      put_line("Ada won't let you assign values"
            & " that are too cold.");
  end;
  begin
    LAB_TEMPERATURE := Type_Convert(32100.0);
    put_line("FAILED to detect too hot.");
  exception
    when CONSTRAINT_ERROR =>
      put_line("Ada won't let you assign values"
            & " that are too hot.");
  end;
end Range_Checking_Example;

-------- When you run it, here's what you get: --------

C:>RANGE_CHECKING_EXAMPLE
Ada will let you assign reasonable values.
Ada won't let you assign values that are too cold.
Ada won't let you assign values that are too hot.
```

2.5. Generic INTEGER_UNITS package

Listing 2 shows the generic INTEGER_UNITS package. Take some time here to read through it before we discuss it.

2.5.1. Generic Parameters

There are three generic parameters. The first is the integer type. The four logical choices for this parameter are integer, Integer_8, Integer_16, and Integer_32. Of these, Integer_32 is the one I generally use. I would use the others only in those applications where 1) the range of values is small enough to be represented by 8 or 16 bits, 2) there is a significant speed advantage associated with a smaller size, and 3) speed is important to that application.

Notice that I didn't make Integer_32 the default data type. That's because Ada doesn't allow a default for a generic type. Notice, too, that I can't instantiate this package for type float because the type is range <>. The range <> must always be replaced with an integer type.

The other two generic parameters are MIN and MAX. I expect that most of the time you will want the full range of values so I made the default range as large as possible. There are times when you may wish to restrict the range, and the MIN and MAX parameters give you the option of doing that.

2.5.2. Type Conversions

There are three type conversion functions named Type_Convert, "+", and "-". They all do the same thing. They all add units to a pure number. That is, they change "5" to "5 feet". Assume that DISTANCE has been declared to be of type Feet, where Feet has been derived from an instantiation of this package, it would not be legal to say DISTANCE := 5;. That's because 5 is a universal_integer, and DISTANCE is derived from a private type. The private type happens to be an integer, but that is irrelevant. Ada sees them as different types, and that's exactly what I want. I want to make a distinction between Feet and pure numbers.

Somehow we have to be able to jump the barrier between Feet and pure numbers. The Type_Convert function does that. We can say DISTANCE := Type_Convert(5); and Ada will know that we mean 5 feet. I usually use the long phrase Type_Convert because I want to make it obvious that I am dimensioning a number. Sometimes, however, that makes a program line awfully long, and distracts from other things in the line that I feel are more important. In those cases I use the "+" or "-" operator to do the same thing. (Note: the Meridian AdaVantage version 2.1 compiler sometimes has trouble differentiating the + type convert from the unary + operator. It generates an error message, which is easily eliminated by replacing the + with Type_Convert.)

There is one other type conversion routine, called Dimensionless. It is the opposite of the three routines we have just discussed because it removes dimensional units instead of adding them. I'm going to delay the discussion of this function for a moment because it will make more sense after we have discussed dimensioned arithmetic.

2.5.3. Dimensioned Arithmetic

Most of the arithmetic functions are self-explanatory. It should be obvious why you need the common operators like addition and subtraction, so let's skip over them. Things don't get interesting until we get down to the multiplication and division operators.

Conspicuously absent is function "*"(LEFT, RIGHT : Units) return Units;. That's because the product of two dimensioned quantities has different dimensions. (5 feet X 2 feet is not 10 feet, it's 10 square feet.) Multiplication is legal only when one of the numbers is a pure (dimensionless) number. (5 times 2 feet = 10 feet.) The pure number can be on the left or right side of the multiplication operator, so there are two definitions of multiplication of a dimensioned quantity times a pure number.

There are three division operators. The first division operator divides a dimensioned quantity by a dimensionless number, yielding a dimensioned result. (For example, 10 feet / 5 = 2 feet.) Ada's predefined division operator for integer types truncates toward zero, rather than rounding. Therefore, 9 feet / 5 = 1 foot.

Unlike multiplication, division isn't symmetrical. Dividing by a dimensioned quantity changes the units. (1_000_000 / 1 Second = 1 Mega_Hertz.) That's why there isn't a mixed division operator with Integer_type on the left and Units on the right, as there is for multiplication.

The second division operator divides two dimensioned quantities (feet/feet, for example) and returns a dimensionless result. Just like Ada's predefined integer division, the result is truncated toward zero rather than rounding.

It is possible that you would like to get an exact ratio. The third division operation does that. The third division operation can be used to divide 10 feet by 3 feet to determine the ratio of the lengths is 3.333 to 1. If the predefined type float doesn't have enough digits of precision for you (which I think is unlikely), the Dimensionless function (described later) can be used to get more precise results.

If you want a division routine that rounds instead of truncating, use the third division routine to get the exact ratio and do an explicit type conversion to an integer type. (Explicit integer type conversions automatically round the result for you.) For example, if X and Y are of type Feet, INT is type integer, and F is type float, you can say F := X/Y; INT := integer(F);.

Rem and mod and all the relational operators work just like you would expect them to, so lets skip down to the Dimensionless function.

2.5.4. Removing Dimensional Units

The Dimensionless function is the inverse of Type_Convert. It converts dimensioned quantities to pure numbers. It should be used with caution because it defeats the strong type checking we have worked so hard to achieve. Normally you will use it inside a special arithmetic function. For example, if you want to write a function that divides objects of type Feet by objects of type Seconds and produces a result in type Feet_per_sec, you will have to use Dimensionless to convert to pure numbers for the intermediate calculations and thenuse Type_Convert to change the result into Feet_per_sec. Figure 7 shows how to do this. The Dimensionless function could also be used to find a high precision ratio, as in Figure 8.

One of the most common uses for the Dimensionless function is for output. You can't instantiate the TEXT_IO package INTEGER_IO for any dimensional data type, such as Feet, because Feet is a private type, not an integer type. (I wouldn't instantiate INTEGER_IO even if I could, but that's a story we will save for the section on user interfaces.) A good way to print a dimensioned integer is shown in Figure 9. An even fancier way is shown in Figure 10. (These examples use the IMAGE attribute, but you will see something even better than the IMAGE attribute in the ASCII_UTILITIES package.)

2.5.5. Generic Bodies

Let's return to Listing 2 again. This is one of the rare instances where I put the package body in the same file as the package specification. Normally I separate the package body from the specification. I can't always do that with generic packages. The Ada language specification expressly allows vendors to require all generic parts (bodies and subunits) to be in a single file, and some vendors have taken advantage of that. That's a real nuisance for programmers, but it apparently makes it much easier for vendors to implement generics.

The package body is trivial, even if lengthy. In general, each function body just converts to the Integer_type, does the operation, and converts the output to the appropriate type.

2.5.6. Instantiation

Generic packages need to be instantiated before they are used. I instantiated the package for 32-bit integers and put it in a library, and called its instantiation DIM_INT_32. You've seen it used in five of the last six figures. Now it is time to see the package itself. Its simple listing is given in Listing 3. It uses integers with 32-bit range regardless of the target computer. Since I did not include values for MIN and MAX, the default values (Integer_32'FIRST and Integer_32'LAST) are used.

Figure 7. Dimensional division.

```
------------------------------------------------------------
-- The Dimensionless function can be used in special
-- operators which convert unit automatically.

with DIM_INT_32, STANDARD_INTEGERS;
procedure Program_Fragment is

  -- create some dimensional data types
  type Feet         is new DIM_INT_32.Units;
  type Feet_per_sec is new DIM_INT_32.Units;
  type Milliseconds is new DIM_INT_32.Units;

  -- define some dimensioned objects
  MOVEMENT, PRESENT_POSITION, PAST_POSITION : Feet;
  SPEED : Feet_per_sec;
  DELTA_T : Milliseconds;

  -- Tell Ada how to divide Feet by Milliseconds
  -- to get an answer in Feet_per_sec (including
  -- the scale factor of 1000.)
  function "/"(LEFT : Feet; RIGHT : Milliseconds)
    return Feet_per_sec is
    use STANDARD_INTEGERS;
    X, Y, Z : Integer_32;
  begin
    X := Dimensionless(LEFT);
    Y := Dimensionless(RIGHT);
    Z := Integer_32(1000.0 * float(X) / float(Y));
    return Type_Convert(Z);
  end "/";

begin
  loop

  -- Missing statements here have assigned values to
  -- PRESENT_POSITION, PAST_POSITION, and DELTA_T.

    -- Ada checks the next two lines for
    -- dimensional consistency, and they are OK.
    MOVEMENT := PRESENT_POSITION - PAST_POSITION;
    SPEED := MOVEMENT / DELTA_T;

  -- Do something with SPEED and PRESENT_POSITION
  -- and exit the loop if appropriate.

    PAST_POSITION := PRESENT_POSITION;
  end loop;
end Program_Fragment;
```

```
Figure 8. Precise division.
---------------------------------------------------------
-- The Dimensionless function can be used to obtain a
-- ridiculously precise ratio.

with DIM_INT_32; use DIM_INT_32;
procedure Program_Fragment is

  type Feet     is new DIM_INT_32.Units;
  type Float_15 is digits 15;

  PRESENT_POSITION, PAST_POSITION : Feet;
  PRECISE_RATIO : Float_15;

begin

  -- Missing statements here have assigned values to
  -- PRESENT_POSITION and PAST_POSITION.

  PRECISE_RATIO := Float_15(Dimensionless(PRESENT_POSITION))
   / Float_15(Dimensionless(PAST_POSITION));

end Program_Fragment;
```

```
Figure 9. Simple output.
------------------------------------------------------------
--The Dimensionless function can be used to output a value.

with DIM_INT_32; use DIM_INT_32;
with STANDARD_INTEGERS; use STANDARD_INTEGERS;
with TEXT_IO; use TEXT_IO;
procedure Program_Fragment is

  type Feet is new DIM_INT_32.Units;

  PRESENT_POSITION : Feet;

begin

  PRESENT_POSITION := Type_Convert(10);
  put("The present position is");
  put(Integer_32'IMAGE(Dimensionless(PRESENT_POSITION)));
  put_line(" feet.");

end Program_Fragment;
```

```
Figure 10. Better output.
------------------------------------------------------------
-- If there are many places in your program where
-- dimensional data types will be printed, it might be worth
-- while to do write a function to do it.

with DIM_INT_32; use DIM_INT_32;
with STANDARD_INTEGERS; use STANDARD_INTEGERS;
with TEXT_IO; use TEXT_IO;
procedure Program_Fragment is

  type Feet is new DIM_INT_32.Units;

  PRESENT_POSITION : Feet;

  procedure put(X : Feet) is
  begin
    put(Integer_32'IMAGE(Dimensionless(X)));
    put(" feet");
  end put;

begin

  PRESENT_POSITION := Type_Convert(10);
  put("The present position is");
  put(PRESENT_POSITION);
  put_line(".");

end Program_Fragment;
```

2.6. Generic FLOAT_UNITS package

The FLOAT_UNITS package, shown in Listing 4, is almost identical to the INTEGER_UNITS package. Therefore most of the comments about the INTEGER_UNITS package hold equally true for the FLOAT_UNITS package. Let's just concentrate on the differences. First, Listing 4 is instantiated for a type described as digits <> instead of range <>. This means we can instantiate it for any real type.

Second, the limits MIN and MAX are real numbers instead of integers. I chose to give MIN and MAX fixed defaults rather than an attribute. I gave serious consideration to letting the defaults be -1.0 * SAFE_LARGE and SAFE_LARGE, and finally decided against it. There was no compelling argument to make a decision one way or another, and I finally picked +/- 1.0e25 as the limits because I felt they will cover most real applications and they are less than the published limits for the type float for all the compilers I use. If you need an exceptionally large range you can always instantiate this package with any limits the compiler can support. Using fixed values instead of attributes guarantees that the default limits will be the same no matter which compiler is used, so it is less likely that there will be an unpleasant surprise when programs are moved from one computer to another.

2.6.1. Division, Remainder, and Modulo Operators

Normally if you take the ratio of two real numbers, you want a real result. There are times, however, when you want the old elementary school definition of division. ("How many times does X go into Y?") For example, "How many complete revolutions has an object made if it has turned 25.89 radians?" You might also want to know the orientation of an object that has turned 25.89 radians. A division operator that returns an integer ratio, and a modulo operator that takes a real argument, would be helpful in converting 25.89 radians to 4 revolutions and 0.76 radians. Ada does not define division of two real numbers with an integer result, nor does she have rem and mod operators for real types. That's not a major problem. It was easily solved.

I started with INTEGER_UNITS package and converted integer types to floating point types. I left the two common division operators (function "/"(LEFT : Units; RIGHT : Float_type) return Units; and function "/"(LEFT, RIGHT : Units) return Float_type;) clustered with the other common arithmetic operators, but simply for cosmetic reasons I moved function "/"(LEFT, RIGHT : Units) return Integer_type; to the end of the list of arithmetic operators, next to the mod and rem. (It put the division operator close to the comment that describes its use.) Then I wrote the bodies of the integer division, rem, and mod operators. (INTEGER_UNITS just used the predefined Ada operators. Since the equivalent operators don't exist for real types, I had to write them myself for FLOAT_UNITS.)

When I first wrote this package I used integer for the result of the integer ratio. That was a bad idea because, as the broken record says, the range of integer depends on the implementation. Then I decided to use STANDARD_INTEGERS.Integer_32. That wasn't a good idea either, because I don't expect ratios to be large and it seems silly to burden a 16 bit computer with double precision arithmetic to calculate a number less than 100.

I finally did the smart thing. I added a fourth generic parameter, Integer_type, and let the programmer make the best choice based on the application.

2.6.2. First and Last Functions

When I talked about the INTEGER_UNITS package I intentionally avoided mentioning the First and Last functions because I wanted to wait until now to discuss them. Both INTEGER_UNITS and FLOAT_UNITS have First and Last functions.

If you have done much Ada programming I'm sure you've found how valuable the FIRST and LAST attributes are. You have probably come to take them for granted. If I didn't provided you the First and Last functions, it would be distressing to discover that a dimensioned data type, such as Celsius, has no FIRST or LAST attributes.

There are two reasons why Ada won't allow you to write a statement like ABSOLUTE_ZERO := Celsius'FIRST;. The first is that even if type Celsius is new DIM_INT_32.Units; then type Celsius is a private type, not an integer type. The fact that type Units is represented by a 32 bit integer doesn't matter. Private types don't have FIRST and LAST attributes no matter how they are represented.

The second reason is that if type Celsius is new DIM_FLOAT.Units; then type Celsius is a private type that is represented by a type (float) that doesn't have FIRST and LAST attributes to begin with!

The First and Last functions provide a way of obtaining the MIN and MAX values used to instantiate the INTEGER_UNITS or FLOAT_UNITS package. They aren't quite as clean as actual attributes, but at least it is possible to declare ABSOLUTE_ZERO : constant Celsius := Type_Convert(TEMPERATURE.First); (regardless of whether TEMPERATURE is an instantiation of INTEGER_UNITS or FLOAT_UNITS).

2.7. DIM_FLOAT package

It was convenient to instantiate the FLOAT_UNITS package for the predefined types float and STANDARD_INTEGERS.Integer_32 and put it in the program library. I called this instantiation DIM_FLOAT. It is found in Listing 5. The Meridian AdaVantage version 2.1 refused to instantiate Integer_type for 32 bit integers, so I use the version shown in Listing 6 on the Meridian compiler.

2.8. Non-existent NUMERIC_UNITS package

The two packages INTEGER_UNITS and FLOAT_UNITS are so similar, it is tempting to ask, "Why not combine both into a single generic package that can be instantiated for either integers or real numbers?" I must admit I was seduced into trying this, but I quickly recognized the error of my ways and escaped from the trap. I'll show you how I started, so you won't get tricked into trying the same thing.

I wrote a generic package called NUMERIC_UNITS for a type described as private. This package could have been instantiated for any numeric type. If I did this, I would have to also supply some arithmetic operators because private types don't come with arithmetic. Part of the package specification is shown in Figure 11.

As it turns out, Figure 11 isn't very practical. Notice that I haven't supplied any default values for MIN and MAX. I can't, because numbers like integer'LAST and 1.0e25 aren't private type values. That means an instantiation would always have to supply the limits. That's a minor inconvenience. The real show-stopper, though, is an ambiguity in the division operators. If I worked long and hard enough, I think I could come up with a way to keep the division operators straight. I'd probably have to replace the "/" operators with distinct functions with names like Float_Result_Division and Integer_Result_Division and instantiate the package very carefully. The only way you can really appreciate the problem is to fool around with it yourself and see what problems you run into.

Suppose you do figure out how to get the universal NUMERIC_UNITS package to work. What have you gained? It won't do anything you couldn't do by instantiating the integer or real versions already discussed.

There is one possible advantage for combining the two package. Whenever there is a good reason to change the INTEGER_UNITS package, the FLOAT_UNITS package should probably be changed for the same reason. It might be easier to maintain the single NUMERIC_UNITS package than to maintain both INTEGER_UNITS and FLOAT_UNITS, so combining the two packages might make maintenance easier.

There are two possible disadvantages. The maintenance considerations discussed in the previous paragraph may really be a disadvantage. It might not be true that changes made to the INTEGER_UNITS package should also be made to the FLOAT_UNITS package, so coupling the two packages might cause bugs or maintenance difficulties. It will certainly be more difficult to instantiate the NUMERIC_UNITS package because of the limit and division operator problems.

The conclusion is that you waste an awful lot of engineering time (that is, money), for a questionable improvement if you combine INTEGER_UNITS and FLOAT_UNITS into a single generic package. The combined package is elegant, but elegance for elegance's sake doesn't make sense in the engineering world. It is better to keep the INTEGER_UNITS and FLOAT_UNITS package separate.

2.9. Non-existent FIXED_UNITS package

Since I have written an INTEGER_UNITS package and a FLOAT_UNITS package it is reasonable to expect me to have written a FIXED_UNITS package. I haven't. There were two reasons why not.

First, the FIXED_UNITS package requires some options that can't easily be selected using generic parameters. Sure, I could use generic parameters for the number of digits, minimum, and maximum values, but there are more decisions that need to be made. For example, sooner or later you will have to convert the value to a character string of many digits. Do you want to separate digits by commas, periods, or underlines? There are even more difficult decisions to be made regarding the location of the decimal point after arithmetic operations. I just couldn't guess what you would want, so I didn't try.

Second, I don't think the FIXED_UNITS package is necessary, and it might not even be a good idea. If I wrote a FIXED_UNITS package and included it with INTEGER_UNITS and FLOAT_UNITS, then it would encourage people to use FIXED_UNITS when they probably should use INTEGER_UNITS or FLOAT_UNITS. I don't like to encourage the use of fixed- point arithmetic.

Figure 11. NUMERIC_UNITS example.

```
------------------------------------------------------------
-- Sometimes trying to make a generic unit too universal is
-- more trouble than it is worth. Here's what happens if you
-- try to combine the integer and floating dimensional
-- packages. (This is just the beginning of trouble.)

with STANDARD_INTEGERS;
generic
  type Numeric_type is private;
  with function "-"(RIGHT : Numeric_type)
    return Numeric_type is <>;
  with function "+"(LEFT, RIGHT : Numeric_type)
    return Numeric_type is <>;
  with function "-"(LEFT, RIGHT : Numeric_type)
    return Numeric_type is <>;
  with function "*"(LEFT, RIGHT : Numeric_type)
    return Numeric_type is <>;
  with function "/"(LEFT, RIGHT : Numeric_type)
    return Numeric_type is <>;
  with function "/"(LEFT, RIGHT : Numeric_type)
    return STANDARD_INTEGERS.Integer_32;
  with function "/"(LEFT, RIGHT : Numeric_type)
    return float;
  with function "abs"(RIGHT : Numeric_type)
    return Numeric_type is <>;
  with function "rem"(LEFT, RIGHT : Numeric_type)
    return Numeric_type is <>;
  with function "mod"(LEFT, RIGHT : Numeric_type)
    return Numeric_type is <>;
  with function ">"(LEFT, RIGHT : Numeric_type)
    return boolean is <>;
  with function ">="(LEFT, RIGHT : Numeric_type)
    return boolean is <>;
  with function "<"(LEFT, RIGHT : Numeric_type)
    return boolean is <>;
  with function "<="(LEFT, RIGHT : Numeric_type)
    return boolean is <>;
package NUMERIC_UNITS is

 type Units is new Numeric_type;

  -- These functions convert pure numbers into
  -- dimensioned quantities, and vice versa.

  function Type_Convert(X : Numeric_type) return Units;
  function "+"(X : Numeric_type) return Units;
  function "-"(X : Numeric_type) return Units;

  -- These are all the arithmetic functions you need.

  function "+"(RIGHT : Units)
    return Units;
  function "-"(RIGHT : Units)
    return Units;

-- and so on

end NUMERIC_UNITS;
```

2.9.1. Problems with Fixed-Point

You might be tempted to use Ada's predefined fixed point type to represent certain kinds of physical quantities, like dollars. That's not a good idea. Suppose you write type Dollars is delta 0.01 range -10_000_000.00 .. 10_000_000.00;. You might be surprised to discover that the accuracy isn't exactly one cent. The LRM allows Dollars'SMALL to be 1/100 or 1/128 (or any other fraction smaller than 1/100). If it uses 1/128, then there could be a few cents roundoff error if a long column of numbers is added.

There is an optional pragma you could use to specify Dollars'SMALL, but it might not be supported on the compiler you want to use. Even if it is, whenever you depend on pragmas to make your program work, you are asking for portability problems.

The LRM gives the vendors so much freedom when it comes to implementing fixed-point types, you can't be sure what you are getting. DEC Ada just uses a 32 bit integer, and interprets the LSB as the delta value. That means DEC Ada could represent the type Dollars described above perfectly, but could not represent Big_Bucks if type Big_Bucks is delta 0.01 range -22_000_000.00 .. 22_000_000.00; because that exceeds the range of values that can be represented by 32 bits.

I have never found a situation where Ada's fixed-point data type was useful. If I need nine or fewer digits of range, with one digit of absolute accuracy, then I can use Integer_32. Just for the sake of argument, suppose there was a situation where I needed 50 digits of range with one digit of accuracy, then the implementation of Ada's fixed-point type probably wouldn't be sufficient to support that many digits anyway. Even if it did, I couldn't be sure I would get the accurate results from it.

2.9.2. Custom Fixed-Point Types

If I did have an application where I needed exceptionally high accuracy, I would write a package that used a private data type to represent the large, fixed-point numbers. It's so easy to do, I usually assign it as a problem in my beginning Ada class. There are three ways students commonly solve the problem.

Most students use the same Binary Coded Decimal (BCD) approach that 4-bit hand-held calculators use. This involves creating an array of digits that represents the number. Arithmetic operators process the numbers one digit at a time, just like you would if you were adding two long numbers using paper and pencil. They use hidden variables with names like CARRY, BORROW, and PARTIAL_PRODUCT to hold some intermediate results. A boolean variable tells if the value is positive or negative, and another integer tells where the decimal point is.

It isn't very efficient to use a 16- or 32-bit computer to process numbers 4 bits at a time, so some students use an integer to represent a group of three or four digits. The technique is basically the same as BCD, but the computer does several digits at a time, so it is more efficient.

Perhaps the most difficult part of these two approaches is converting arrays of digits to character strings, and vice versa. This sometimes leads a third group of students to use a character string as the private type representing the long number. They still process the data one digit at a time, but instead of arrays of integers they use arrays of characters and define special operations that work on characters.

2.9.3. FIXED_UNITS exercise

If you like, you can try to write a FIXED_UNITS package. Start with the INTEGER_UNITS specification and make a few modifications. (You don't have to make it generic if you don't want to.) You will have to replace the definition of the private type with something that can represent many digits. You can use one of the three suggested approaches above, or come up with something totally different. You will probably want to replace the Type_Convert and Dimensionless functions with Image and Value functions (similar to those found in the ASCII_UTILITIES package in the next section), because you can't use numeric types to represent all the values you want to represent. (If you could, you wouldn't need the FIXED_UNITS package.)

If you want to be able to multiply a monetary figure (which has two decimal places) by 6.5% sales tax (three decimal places), then your private type will have to have a "floating decimal point" so it can represent numbers to two or three (or any other number) or decimal places.

If you allow a floating decimal point it raises some interesting problems. For example, If you subtract 0.123 from 0.9, you will have to move the decimal point before you find the difference. Once you have performed the subtraction, should the answer be 0.777 or 0.8?

The problems with fixed-point arithmetic make it more trouble than it is worth. I think you would be better off to avoid fixed-point arithmetic whenever you can.

2.9.4. Fixed-Point Computers

Ten or fifteen years ago, before floating-point coprocessors were common, some computers were designed to use fixed-point binary numbers. A fixed-point computer does surprising things if you don't know how it works. For example, a computer that uses integer arithmetic will multiply 2#0100# times 2#0010# and yield the expected result 2#01000# (4 * 2 = 8). When a fixed-point computer multiplies 2#0100# times 2#0010# the answer is 2#00001# (1/4 * 1/8 = 1/32). That's because integer machines assume a binary point to the right of all the bits, while a fixed-point machine assumes a binary point to the left of all the bits.

Fixed-point computers are no longer in vogue, but they are still in some military systems. An Ada compiler for a fixed-point target computer might take advantage of certain machine features if you declare real data types to be fixed- point types with deltas that are a power of 2. I haven't used an Ada compiler for a fixed-point computer, so I don't know what it does, but you may want to look into it you find yourself in that situation.

2.10. TRIG package

The dimensioned numbers we have developed can be put to good use in a math package I call TRIG. The TRIG package specification shown in Listing 7 gives your Ada programs new levels of reliability and transportability.

It is more reliable because it is impossible to provide the wrong argument to a trigonometric function. I'm sure any reader who has written programs involving Sines and Cosines has, at least once, tried to take the sine of an angle expressed in degrees using a function expecting an input in radians, or vice-versa. An even more common mistake is multiplying by PI_OVER_180 when the angle should have been divided by that factor, or vice-versa. Those days are gone forever if you use the TRIG package.

2.10.1. Type Naming Convention

The TRIG package creates two data types, Deg and Rad, which you can use for angular variables. I could have called these data types Degrees and Radians, but that violates a simple convention I use when deriving dimensional units. Whenever I derive a dimensional unit from INTEGER_UNITS, I spell the units out completely. If the dimensional unit is derived from FLOAT_UNITS, then I use an abbreviation. Therefore, I know immediately that type Feet is an integer data type, but type Ft uses real numbers. This means I can tell its range and precision without having to search through the code to find out if the unit was derived from INTEGER_UNITS.Units or FLOAT_UNITS.Units. The convention is easy to remember because both abbreviations and real numbers have dots in them. Since the TRIG package uses real numbers, rather than integers, I used Deg and Rad for the type names.

2.10.2. Overloaded Function Names

Notice there are two overloaded versions of Sin. That is, there are two different functions with the same name. One Sin function works for type Rad and the other works for type Deg. When you declare an angular object you assign its dimensions by making it type Rad or Deg. When you call for the Sine of that angle, Ada will automatically select the correct Sin routine from the TRIG package based on the data type. When you take an inverse function, the TRIG package will automatically return degrees or radians, whichever is correct. If ANGLE_1 and ANGLE_2 are expressed in different units, you can always say ANGLE_1 := TRIG.Convert_Units(ANGLE_2); and you will get the correct result, no matter which one is in degrees and which one is in radians. (If ANGLE_1 and ANGLE_2 already are the same kind of units, both degrees or both radians, Ada will tell you so at compiler time with a fatal error message.) You won't need any more PI_OVER_180 constants cluttering up your program.

2.10.3. Portability

The TRIG package also makes your programs more portable because it can be used as a standard math package. You were probably surprised when you discovered Ada has no predefined math package. That may seem strange for a military programming language, but it makes good sense. Since Ada is strongly typed you would have to have separate math function for every real data type. Since the number of distinct real data types is unbounded, that would be a lot of math routines. A standard generic math package could be instantiated for each real data type (the way FLOAT_IO is), but that doesn't completely solve the problem. Some machines have an optional math coprocessor. Ada would have to have multiple versions of the math library, some of which would use the coprocessor and others wouldn't. Furthermore, some applications may need immediate answers good to three decimal places, while other applications need higher precision and have all the time in the world to compute the answer. If Ada had a built-in standard math library, a trade-off would have to be made, and the resulting math routines would be good for some applications but not others.

These are all good reasons for leaving the math routines out of the Ada language. I wouldn't have it any other way. Compiler vendors, however, wisely recognized that most of us programmers use math functions often in our application programs. Furthermore, we want to solve the problems we are being paid to solve, not reinvent the Sine function. We would abuse their sales representatives if they didn't include a math library as part of the programming environment, so most compilers today come with a math library. You will probably find a package called something like MATH_LIB or GENERAL_MATH in the system library.

Math libraries often cause portability problems. They generally have different names for the library as well as the functions. For example, the square root function might be called Sqr or Sqrt. Some of math libraries are generics and some have already been instantiated. Some math libraries have two functions, Sin and Sind, for taking sines in radians and degrees. Other libraries only have a radian sine function. Suppose your program used Sind to find the sine of angles expressed in degrees, and you transported it to another system that used Sind for a double-precision sine in radians. How long would it take you to find that bug?

I run into these problems a lot because I use so many different Ada compilers. I solved the problem by using the TRIG package as a shell over the underlying math library. Once I get the TRIG package ported to a new system, then all my other programs can be transported without any math modifications.

The TRIG package specification remains the same on every system, but the body has to be specially tailored for each system. Listing 8 shows the TRIG package body for the DEC compiler. It simply calls the corresponding VAX/VMS math library routine for each function. The TRIG package body for the

Meridian compiler in Listing 9 uses a slightly different approach, just for the sake of illustrating a different way of doing it. (Mathematicians know the method I used in the Meridian body is inferior, and we will talk about that much later when I show you how the routine was tested.) I pretended that there wasn't any Cos or Tan function available and derived them from the Sin function. (If you are using an embedded computer, you might just have a sine lookup table, or a coprocessor that only computes sines.) In the fall of 1987, Alsys did not officially provide a math library. The Version 3.2 distribution disk contained a math library contributed by a customer. Listing 10 uses that unofficial Alsys math library. No matter what the package body implementation is, the package specification doesn't change, and that makes all the programs that use the TRIG package portable.

2.10.4. Reciprocal Functions

When I wrote this package I had to decide what to put in, and what to leave out. I chose to leave out the reciprocal functions (secant, cosecant, and cotangent). I can't remember the last time I used one of these, so it didn't seem worth the memory space to put them in. If I had put them in, they probably would have just returned 1/Sin, 1/Cos, or 1/Tan. If the application program used the Secant in an equation like X := Y*Secant(THETA);, it would compute X := Y*(1/Sin(THETA); which involves an extra operation and an extra function call (unless the optimizer takes them out). It would be better to simply write X := Y/Sin(THETA);. (Dividing by Secant(THETA) would be even more foolish.) I decided it was better to leave out the reciprocal functions than be tempted to use them.

This TRIG package may not exactly fit your needs, but it is built in such a way that you can modify it to do exactly what you want. You can derived Deg and Rad from any floating point type you desire. You can change the body to use any desired algorithm. Since it isn't part of the language you are free to do whatever you want.

2.10.5. Special Cases

I made some decisions concerning special cases, and those decisions might not be appropriate for your application. Since the TRIG body is given in source code, you are free to change it however you like.

The first special case is the tangent of 90 degrees. The theoretical value is infinite. DEC Ada raises FLOAT_MATH_LIB.FLOOVEMAT. I could have handled this by simply raising NUMERIC_ERROR in its place and let the application program figure out how to handle it, but instead I decided to return a very big number. The problem is, how big is "very big?" I decided that since I had previously decided limits for DIM_FLOAT, I might as well use the same maximum and minimum here. I return DIM_FLOAT.Last for the Tangent of 90 degrees, and DIM_FLOAT.First for the Tangent of 270 degrees. If that's not appropriate for your application, feel free to change it.

The second special case is the two argument arctangent where both arguments are zero. What is the bearing from the origin to the origin? It is undefined. You could say that it is a stupid question that doesn't deserved to be answered, but consider this: People who are trying to drop bombs on you generally try to fly right over your head, and sometimes they succeed. When that happens the elevation is 90 degrees and the azimuth is Atan(0.0,0.0). It could be an important moment in your life, and you want the right answer. If you are flying an airplane and pull a vertical loop, there are moments when you are flying straight up or straight down, so your heading is Atan(0.0,0.0).

My first approach to the problem was to define Atan(0.0,0.0) to be 0.0 because I wanted to avoid raising an exception. That drove an aircraft simulation nuts when it tried to simulate a loop if the aircraft wasn't flying due north or due south. I decided it was better to raise the INVALID_ARGUMENT exception and let the application program handle it. (The application program handled it by using the previous heading, and that worked fine.) Your situation may require a different solution, and you are free to do whatever you like with the source code.

2.10.6. Which Way is Up?

The four quadrant arctangent function is a common cause of programming errors because it is easy to confuse the arguments. To help you appreciate the problem, let me try to confuse you.

Suppose a position is expressed in polar coordinates. The distance from the origin is 10 and the angle is 30 degrees. If we need to convert this to cartesian coordinates, we compute X = 10 * Cos(30) = 8.66. Then we find Y = 10 * Sin(30) = 5.00. No problem. But suppose we want to convert that point back to polar coordinates. Do we use Atan(8.66,5.00) or Atan(5.00,8.66)?

The common convention is that Atan(A1, A2) returns the arctangent of A1/A2, so the first argument

```
Figure 12. Polar Coordinates: Mathematicians measure angles counterclockwise
from the horzontal axis.
-----------------------------------------------------------------

[This graphic figure can't be produced using text characters
alone.  You must draw a line from the origin to the '*' character
at point (8.66, 5.00).  Then draw an arc from the X-axis to the
line, next to the "30 Degrees" label.]


 10 +--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
    |                             |                             |
 9  +  +  +  +  +  +  +  +  +  +  +  +  +  +  +  +  +  +  +  +  +
    |                             |                             |
 8  +  +  +  +  +  +  +  +  +  +  +  +  +  +  +  +  +  +  +  +  +
    |                             |                             |
 7  +  +  +  +  +  +  +  +  +  +  +  +  +  +  +  +  +  +  +  +  +
    |                             |                             |
 6  +  +  +  +  +  +  +  +  +  +  +  +  +  +  +  +  +  +  +  +  +
    |                             |                 (8.66,5.00) |
 5  +  +  +  +  +  +  +  +  +  +  +  +  +  +  +  +  +  +  + *+  +
    |                             |                             |
 4  +  +  +  +  +  +  +  +  +  +  +  +  +  +  +  +  +  +  +  +  +
    |                             |                             |
 3  +  +  +  +  +  +  +  +  +  +  +  +  +  +  +  +  +  +  +  +  +
    |                             |                             |
 2  +  +  +  +  +  +  +  +  +  +  +  +  +  +  +  +  +  +  +  +  +
    |                             |                             |
 1  +  +  +  +  +  +  +  +  +  +  +  +  +  +  +  +  +  +  +  +  +
    |                             |               30 Degrees    |
 0  +--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
    |                             |                             |
-1  +  +  +  +  +  +  +  +  +  +  +  +  +  +  +  +  +  +  +  +  +
    |                             |                             |
-2  +  +  +  +  +  +  +  +  +  +  +  +  +  +  +  +  +  +  +  +  +
    |                             |                             |
-3  +  +  +  +  +  +  +  +  +  +  +  +  +  +  +  +  +  +  +  +  +
    |                             |                             |
-4  +  +  +  +  +  +  +  +  +  +  +  +  +  +  +  +  +  +  +  +  +
    |                             |                             |
-5  +  +  +  +  +  +  +  +  +  +  +  +  +  +  +  +  +  +  +  +  +
    |                             |                             |
-6  +  +  +  +  +  +  +  +  +  +  +  +  +  +  +  +  +  +  +  +  +
    |                             |                             |
-7  +  +  +  +  +  +  +  +  +  +  +  +  +  +  +  +  +  +  +  +  +
    |                             |                             |
-8  +  +  +  +  +  +  +  +  +  +  +  +  +  +  +  +  +  +  +  +  +
    |                             |                             |
-9  +  +  +  +  +  +  +  +  +  +  +  +  +  +  +  +  +  +  +  +  +
    |                             |                             |
-10 +--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
   -10 -9 -8 -7 -6 -5 -4 -3 -2 -1  0  1  2  3  4  5  6  7  8  9 10
```

should be the Y component and the second argument should be the X component. So, if the X coordinate is 8.66 and the Y coordinate is 5.00, the correct angle is found by computing Atan(5.00,8.66).

Now, what is the bearing of a target 8.66 miles East and 5.00 miles North? You think it is Atan(5.00,8.66)? Think again! Atan(5.00,8.66) yields 30 degrees, but the correct answer is 60 degrees.

Mathematicians compute the angle of a point in polar coordinates counterclockwise from the horizontal axis, as shown in Figure 12. Anyone who has ever used a compass (especially pilots and radar operators) knows 0 degrees is North and 90 degrees is East. Angles are measured clockwise from the vertical axis, as shown in Figure 13. Therefore, the bearing to a target 8.66 miles East and 5.00 miles North is given by Atan(8.66,5.00).

```
Figure 13. Directions: Pilots measure angles clockwise from North.
-----------------------------------------------------------------

[This graphic figure can't be produced using text characters
alone.  You must draw a line from the origin to the '*' character
at point (8.66, 5.00).  Then draw an arc from the Y-axis to the
line, next to the "60 Degrees" label.]


 10 +--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
    |                             |                             |
 9  +  +  +  +  +  +  +  +  +  +  +  +  +  +  +  +  +  +  +  +  +
    |                             |                             |
 8  +  +  +  +  +  +  +  +  +  +  +  +  +  +  +  +  +  +  +  +  +
    |                             |                             |
 7  +  +  +  +  +  +  +  +  +  +  +  +  +  +  +  +  +  +  +  +  +
    |                             |                             |
 6  +  +  +  +  +  +  +  +  +  +  +  +  +  +  +  +  +  +  +  +  +
    |                             |                 (8.66,5.00) |
 5  +  +  +  +  +  +  +  +  +  +  +  +  +  +  +  +  +  +  + *+  +
    |                             | 60 Degrees                  |
 4  +  +  +  +  +  +  +  +  +  +  +  +  +  +  +  +  +  +  +  +  +
    |                             |                             |
 3  +  +  +  +  +  +  +  +  +  +  +  +  +  +  +  +  +  +  +  +  +
    |                             |                             |
 2  +  +  +  +  +  +  +  +  +  +  +  +  +  +  +  +  +  +  +  +  +
    |                             |                             |
 1  +  +  +  +  +  +  +  +  +  +  +  +  +  +  +  +  +  +  +  +  +
    |                             |                             |
 0  +--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
    |                             |                             |
-1  +  +  +  +  +  +  +  +  +  +  +  +  +  +  +  +  +  +  +  +  +
    |                             |                             |
-2  +  +  +  +  +  +  +  +  +  +  +  +  +  +  +  +  +  +  +  +  +
    |                             |                             |
-3  +  +  +  +  +  +  +  +  +  +  +  +  +  +  +  +  +  +  +  +  +
    |                             |                             |
-4  +  +  +  +  +  +  +  +  +  +  +  +  +  +  +  +  +  +  +  +  +
    |                             |                             |
-5  +  +  +  +  +  +  +  +  +  +  +  +  +  +  +  +  +  +  +  +  +
    |                             |                             |
-6  +  +  +  +  +  +  +  +  +  +  +  +  +  +  +  +  +  +  +  +  +
    |                             |                             |
-7  +  +  +  +  +  +  +  +  +  +  +  +  +  +  +  +  +  +  +  +  +
    |                             |                             |
-8  +  +  +  +  +  +  +  +  +  +  +  +  +  +  +  +  +  +  +  +  +
    |                             |                             |
-9  +  +  +  +  +  +  +  +  +  +  +  +  +  +  +  +  +  +  +  +  +
    |                             |                             |
-10 +--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
  -10 -9 -8 -7 -6 -5 -4 -3 -2 -1  0  1  2  3  4  5  6  7  8  9 10
```

That why I called the arguments to the TRIG.Atan function EAST_OR_Y and NORTH_OR_X. I makes it easier for me to remember the correct parameter associations.

Things get even worse in three dimensions because there are so many right-handed coordinate systems. Mathematicians like to use X (right), Y (ahead), and Z (up) vectors. Airborne systems like to use NORTH, EAST, DOWN. Ground-based systems like to use EAST, NORTH, UP. Body-referenced coordinates can be X',Y',Z' or X",Y",Z" where the major body axis could be aligned with any of those vectors, depending on the programmer's whim.

Fortunately Ada allows you to define enumeration types that specify body directions in meaningful terms. If you want to use an array to represent a three dimensional velocity in body coordinates you can do this:

```
     type Body_Coordinates is (BOW, STARBOARD, KEEL);
     type Velocities is array(Body_Coordinates) of Feet_per_second;
     MISSILE_SPEED : Velocities;

     MISSILE_SPEED(BOW) := (some_value);
```

You don't have to use arrays to represent three dimensional quantities. For example, you could use a record to represent the attitude of the missile like this:

```
     type Attitudes is
          record
               PITCH, YAW, ROLL : Degrees;
          end record;

     MISSILE_ORIENTATION  : Attitudes;
     MISSILE_ORIENTATION.PITCH := (some_value);
```

In any case, you can avoid confusion by avoiding the use of the ambiguous X, Y, Z labels for each axis.

2.11. COORDINATES package

This chapter began by looking at a poor specification for coordinate transformation package. It is fitting to end the section by applying all we've learned to make it better. Figure 14 shows an improved version of the package specification.

Most of the improvements will show up in the package body, so there isn't too much difference between Figure 14 and Figure 1. The comments telling us that linear distances are measured in feet and angles are in degrees have been eliminated because they say the same things that the component declarations say. (Ironically some code quality metric tools might tell us that the original version is better than the improved version because the original version has more comments.) Replacing the comments with dimensioned data types may not make much difference to human readers, but it makes a big difference to Ada because she can read data types, and she can't read comments. This lets her check dimensional consistency. On a large program she is more likely to find errors than a human would.

2.11.1. Comments

Adding comments to an Ada program can counter-productive, so it is worth while to talk about how to comment an Ada program. Many of you have been taught that the more comments a program has, the better it is. That just isn't true. For example, if your program includes the statement X := A + B; and you comment that line with the phrase "-- Add A to B to get X" you have made the program the program worse, not better.

What's wrong with adding a comment like that? Two things are wrong. First, you have repeated information. Whenever you do that, you create an opportunity for contradiction and confusion. Suppose the program doesn't work properly because a third term, C, has to be added to X. Someone discovers the error and changes the code, but forgets to change the comment. Then when another maintenance programmer looks at the code to solve another problem, he will get confused because the comment contradicts the code. He may figure the comment is correct, and take C out of the equation.

Second, too many frivolous comments can clutter your program so much that you can't easily find the important comments. Once a bad comment is in a program it enjoys all the respect and status of any other comment in the program. There is no easy way to tell good comments from bad comments. I've occasionally used a text editor to remove all the comments from someone else's program because there were so many frivolous comments I couldn't easily find the executable lines of code. In so doing, I probably lost some valuable information, but that information was already lost in the clutter.

Since it is practically impossible to remove bad comments from a program, the only solution to the problem is to not insert them in the first place. Every organization seems to have some internal standard for a header block of comments. These header blocks include things such as the module name, a brief description of what it does, who wrote it, when the last revision was made, and so on. I have my own format which you have seen in the listings, but I don't have any burden for a particular header format. Header blocks aren't a problem. They generally contain important information, and since they are at the beginning of the program they can easily be skipped if they don't. Use whatever style header you like. The comments sprinkled through the listing are the ones that are likely to cause problems.

The nature of the Ada language makes many comments unnecessary. For example, it is important to know which modules depend on each other. In some languages it might be appropriate to include comments telling about these dependencies. This doesn't make sense in Ada because the dependencies are shown by context clauses (i.e. with LIBRARY_UNIT;) and separate clauses (separate(Main_Program)). Ada requires these clauses to be at the beginning of every compilation unit, so you don't have to search through the code to find them. If you refrain from excessive USE clauses, then dot notation will make dependencies obvious in the executable code. For example, if your program is calling the procedure

Figure 14. COORDINATES package specification.

```
-----------------------------------------------------
-- This is an improved version of the POOR_COORDINATES
-- package given in Figure 1.

with DIM_INT_32, TRIG;
package COORDINATES is

  type Feet is new DIM_INT_32.Units;

  type Rectangular_points is
    record
      NORTH : Feet;
      EAST  : Feet;
    end record;

  type Polar_points is
    record
      R     : Feet;
      THETA : TRIG.Deg;
    end record;

  function Transform(RP : Rectangular_points)
    return Polar_points;

  function Transform(PP : Polar_points)
    return Rectangular_points;

end COORDINATES;
```

Normalize from the package LIBRARY_UNIT, your statement will be LIBRARY_UNIT.Normalize(X); and there is no need to comment that Normalize is found in LIBRARY_UNIT.

One thing Ada doesn't do is force you to include information about source file names. This makes it possible for you to write a body stub like, procedure Accumulate is separate; which doesn't tell the name of the file containing the source code for Accumulate. Ada doesn't care what the name of the source file is. Some Ada implementations may record that information in the library as a courtesy to you, so you can write automatic recompilation tools, but it isn't required by the language. Even if the information is in the library, you shouldn't force the maintenance programmer to run a library utility to find it. Always follow a separate clause with a comment like -- File ACCUM.ADA so people will know where to find the source code.

Ada package specifications provide a top level view of what services the package provides without a hint at how it works. The comments should be consistent with this policy. Comments describing the implementation of subprograms have no place in a package specification. I like to let subprogram declarations serve as headings for a paragraph or two describing what the subprogram does. Sometimes I include samples of how the subprogram should be used. If it can raise an exception it is important to include comments telling that the subprogram may raise that exception, and under what circumstances that might happen. When a package declares user-defined exceptions, I always include comments below the exception declaration that tell what subprograms can raise the exception. I try not to repeat what conditions cause that exception to be raised because that information should be in the comments describing the subprogram. (If many subprograms can raise the same exception, I sometimes save space by describing the conditions that cause the exception to be raised under the exception, and let the subprogram specifications refer to the comments in the exception handler.)

Unlike package specifications, Ada package bodies shouldn't include descriptions of what the subprograms do. At best, those comments would exactly duplicate the comments in the package specification. At worst they would contradict those comments and confuse the reader. Most often they would have no effect because a rational person wouldn't think to look in a package body for comments describing the purpose of a subprogram. Package bodies should contain comments describing how the subprograms work. These comments shouldn't describe what you are doing because that should be obvious from the code. If you find you need to comment a line like P := GI - E; you should rewrite it as PROFIT := GROSS_INCOME - EXPENSES; to eliminate the need for the comment. Never write a comment that could just as well be expressed as executable code.

Comments should describe why you are doing something. For example, if the package body says for PORT in 1..MAX_PORTS loop ... the comment should not say, "-- Do the following statements for every input port". Instead, just before the loop there should be a comment saying, "--We need to check all the input ports to see if any new messages have been received." This tells the reader why the loop is there and what it is trying to accomplish, rather than the obvious fact it is a loop.

Good programmers always indent if statements and control loops to show structure. They recognize how white space can improve readability. Blank lines can also do the same thing. If a group of three to ten lines cooperate to do a particular operation, use blank lines to separate those statements from the surrounding code. It has the same effect as breaking an essay into paragraphs. When you have isolated a few statements into a paragraph, it usually is appropriate to begin that paragraph with a comment to serve as a topic sentence. This not only helps the maintenance programmer understand your program, it helps you write better code because you will occasionally discover that one of the statements in the paragraph doesn't fit with the topic sentence. This is usually an indication that it belongs someplace else.

Ada will check everything in your program except the comments. That means you have to pay particular attention to them yourself. Be sure that you don't have a lot of useless comments cluttering up your code.

```
Figure 15. POOR_COORDINATES body.
------------------------------------------------------------
-- The body of the much-maligned package in Figure 1.

with MATH_LIB; -- Meridian Ada utility package
package body POOR_COORDINATES is

  PI : constant float := MATH_LIB.PI;

  function Transform(RP : Rectangular_points)
    return Polar_points is
    RADIANS_TO_DEGREES : constant float
      := 180.0 / PI;
    X : Polar_points;
    R_SQUARED : integer;
    ANGLE, NORTH, EAST : float;
  begin
    R_SQUARED := RP.NORTH * RP.NORTH + RP.EAST * RP.EAST;
    X.R := integer(MATH_LIB.Sqrt(float(R_SQUARED)));
    if RP.NORTH = 0 then
      if RP.EAST >= 0 then
        X.THETA := 90.0;
      else
        X.THETA := 270.0;
      end if;
      return X;
    end if;
    NORTH := float(RP.NORTH);
    EAST  := float(RP.EAST);
    if NORTH > 0.0 and EAST >= 0.0 then -- 0 to 90 degrees
      ANGLE := MATH_LIB.Atan(EAST/NORTH);
    elsif NORTH < 0.0 and EAST >= 0.0 then -- 90 to 180 deg
      ANGLE := PI - MATH_LIB.Atan(EAST/(-NORTH));
    elsif NORTH < 0.0 and EAST < 0.0 then -- 180 to 270 deg
      ANGLE := PI + MATH_LIB.Atan(EAST/NORTH);
    elsif NORTH > 0.0 and EAST < 0.0 then -- 270 to 360 deg
      ANGLE := 2.0 * PI - MATH_LIB.Atan(-EAST/NORTH);
    end if;
    X.THETA := ANGLE * RADIANS_TO_DEGREES;
    return X;
  exception
    when NUMERIC_ERROR =>
      if EAST >= 0.0 then
        X.THETA := 90.0;
      else
        X.THETA := 270.0;
      end if;
      return X;
  end Transform;

  function Transform(PP : Polar_points)
    return Rectangular_points is
    DEGREES_TO_RADIANS : constant float
      := PI / 180.0;
    ANGLE : float;
    X : Rectangular_points;
  begin
    ANGLE := PP.THETA * DEGREES_TO_RADIANS;
    X.NORTH := integer(float(PP.R) * MATH_LIB.Cos(ANGLE));
    X.EAST  := integer(float(PP.R) * MATH_LIB.Sin(ANGLE));
    return X;
  end Transform;

end POOR_COORDINATES;
```

Don't use a comment to repeat something that is expressed in Ada statements. Wisely use comments to point to important information, like the name of the file containing a subunit or the names of all the subprograms that can raise a user-defined exception. Use comments near a subprogram declaration to tell what that subprogram does. Use comments inside a subprogram body to describe why it is doing what it is doing. Use comments and white space inside a subprogram body to break it into logical processing steps. If you need comments inside a subprogram body to explain how it works, you need to ask yourself why that isn't obvious from the code, and perhaps rewrite the code to make the comment unnecessary.

2.11.2. Other Improvements in COORDINATES

The derivation of type Feet forces the distances to be expressed in 32-bit integers (or longer integers with constraints limiting them to 32-bit range). The original version only told us that distances were expressed as integers, but did not fix their length. It used different lengths on different systems. The improved version is more consistent.

The real differences show up in the package bodies. Figure 15 shows the poor version. It works, but it has some problems. Most of the problems have to do with transportability. There is the previously mentioned problem of undefined integer size, and also a reliance on a package found only in the Meridian Ada compiler library. Fortunately Figure 15 has used dot notation rather than a USE clause so a text editor can search for MATH_LIB to find all the potentially nonportable statements.

The original package is cluttered with conversion constants (RADIANS_TO_DEGREES and DEGREES_TO_RADIANS). These are places where errors are likely to creep into the code. It also includes logic to figure out which quadrant to use. At best, this logic was copied out of one of several other application programs that needed a four quadrant arctangent. At worst it was written from scratch and

Figure 16. COORDINATES body.

```
-------------------------------------------------------------
-- A better way to write the package body.

with STANDARD_INTEGERS; use STANDARD_INTEGERS;
package body COORDINATES is

  function Transform(RP : Rectangular_points)
    return Polar_points is
    X : Polar_points;
    R_SQUARED, NORTH, EAST : float;
  begin
    -- begin dimensionless processing
    NORTH := float(Dimensionless(RP.NORTH));
    EAST  := float(Dimensionless(RP.EAST));
    R_SQUARED := NORTH * NORTH + EAST * EAST;
    X.R := Type_Convert(Integer_32(TRIG.Sqrt(R_SQUARED)));
    -- end dimensionless processing
    X.THETA := TRIG.Atan(NORTH,EAST);
    return X;
  end Transform;

  function Transform(PP : Polar_points)
    return Rectangular_points is
    X : Rectangular_points;
    DISTANCE : float;
  begin
    -- begin partially dimensionless processing
    DISTANCE := float(Dimensionless(PP.R));
    X.NORTH := Type_Convert(Integer_32(DISTANCE
                 * TRIG.Cos(PP.THETA)));
    X.EAST := Type_Convert(Integer_32(DISTANCE
                * TRIG.Sin(PP.THETA)));
    -- end partially dimensionless processing
    return X;
  end Transform;

end COORDINATES;
```

took several iterations to get all the bugs out. In either case it took some duplication of effort.

Figure 15 has some nasty surprises in it. First, it blows up for points more than 8.78 miles from the origin. Although 32 bits can almost represent the number of feet in a round trip to the moon, 32 bits can only represent 46,340 feet squared. The declaration R_SQUARED : integer; is inadequate even on 32-bit machines. Second, the four quadrant logic returns an angle of 270 degrees when NORTH and EAST are both zero, instead of raising an INVALID_ARGUMENT exception.

The improved version in Figure 16 is much shorter and cleaner. It uses 32-bit integers no matter what computer it is running on, and it is directly transportable to any system that has the TRIG package on it. It is shorter because the four quadrant arctangent is in the TRIG package (where it belongs). The reuse of TRIG.Atan saves time and improves reliability because the four quadrant logic does not need to be developed and tested again.

In a large program there might be several times when distances are multiplied to compute an area. Then I would write a function that multiplies two distances in Feet and returns a floating-point type Sq_ft. I would also write a function that takes the square root of an object of type Sq_ft and returns Feet. In this little example, distances are only squared once, so it isn't worth the trouble to write a special function. Instead I move into the dimensionless number domain to calculate the square root of the sum of the squares. Since Ada can't check these statements for me, I bracket them with comments to draw my attention to them.

The final version of the COORDINATES package gives us a library unit that defines all the data types the application program is likely to need. It defines Feet for linear distances, Rectangular_points for Cartesian coordinates, and Polar_points for polar coordinates. (A realistic version of this package would use three dimensional coordinates and include more dimensional data types, too.)

This package specification should be given careful consideration, because most of the other modules in the application program will depend upon this package for type definitions. If you change COORDINATES you will have to recompile every module that depends on it. That's a two edged sword. You won't want to add data types to the package because you cut yourself every time you do, suffering the pain of recompilation with every change. But the sword also protects you because it tells you which modules need to be reexamined in light of your recent change in type definitions.

2.11.3. USE Clauses

I avoid USE clauses whenever practical. Special emphasis should be placed on the word "practical" because it was intentionally chosen instead of "possible." It is always possible to avoid a USE clause, but I feel there are times when it isn't practical.

I like to avoid USE clauses because I like to remind myself which package defined the type or subprogram. It shows me exactly were the dependencies are. My general rule is, don't use the USE.

There are exceptions to the rule. Some packages, like TEXT_IO, are so common that I don't need to be reminded about them. A statement like TEXT_IO.new_line; is no more informative than new_line;, so I commonly use TEXT_IO;. But, if I have added a line like TEXT_IO.put_line("Section 1 entered"); for diagnostic purposes, then I retain the dot notation to make it easy to take the diagnostic line out.

Another package that is so common it doesn't need dot notation is STANDARD_INTEGERS. There is real motivation to use STANDARD_INTEGERS; because it makes arithmetic visible. That is, it is legal to say X := Y + Z; if X, Y, and Z are all STANDARD_INTEGERS.Integer_32. If you don't use a USE clause, you have to say X := STANDARD_INTEGERS."+"(Y,Z);. I think that distracts from the program logic, especially if it is part of a complex equation. True, you could use renaming declarations to rename all the arithmetic functions, but that is longer and no clearer than this USE clause and comment:

```
use STANDARD_INTEGERS;
-- makes 32 bit arithmetic operators visible
```

Chapter 3. I/O UTILITIES

One of Ada's unusual features is that she has no I/O instructions. That's because Ada was originally designed to be a language for embedded computers. Embedded computers don't typically have the same kind of peripherals as general purpose computers do. The output peripherals on embedded computers are usually things like missile control surfaces and high explosives. Inputs come from infrared seekers, doppler filter banks, and push buttons. If Ada had the usual I/O services it would be necessary to design a warhead with the same software interface as a card punch.

Since it is impossible to predict appropriate interfaces for all the bizarre I/O devices Ada is likely to use, she was given language features that make it possible to design custom I/O services. Packages, tasking, and the low level interfaces described in MIL-STD-1815A chapter 13, can be used to interface Ada to anything you can imagine.

Embedded computer applications are so unique, it is unlikely I could write reusable embedded computer I/O routines with broad appeal. General purpose applications, on the other hand, often involve an ASCII interface to a CRT terminal. This section, therefore, contains some I/O routines I think you will find useful in general purpose applications. Although these packages don't often apply directly to embedded computer applications, the lessons learned from this section do.

3.1. ASCII_UTILITIES package

I/O often involves ASCII data. File names are specified by ASCII strings. Numbers are displayed as a string of ASCII characters. Users enter a sequence of ASCII digits that must be converted to numbers. The ASCII_UTILITIES package (Listings 11 through 16) contains some handy routines that manipulate ASCII data.

3.1.1. IMAGE Attribute

Ada has a predefined attribute called IMAGE, which takes a discrete value (an integer or an enumeration value) and converts it to an ASCII character string. It simplifies I/O operations because you can put(integer'IMAGE(X)); to output the integer X, without having to instantiate INTEGER_IO.

I often find it annoying that the IMAGE attribute for integers adds a single blank space on the beginning of the string when the number is positive. This causes problems when I try to insert numbers in the middle of a line of text. If I leave a space before the number, then it appears that there were two spaces in front of positive numbers. If I don't leave a space, then negative numbers appear to be hyphenated to the word immediately before it.

The IMAGE attribute for integers also has the sometimes annoying characteristic of returning a string of unpredictable length. When it converts a number to a character string, it uses only as many characters as it needs. This is often desirable because it is frequently necessary to insert a number in the middle of a line of text, and it would look funny if there were lots of leading spaces. There are other times, though, when you want the number to be converted to a string with a certain number of characters regardless of the value of the number. For example, when converting the internal representation of 14 February, 1988, to the 8 character string "02/14/88" for output, you want 2 characters each for the month, day, and year. Or you might want to give the file containing the data from flight number 375 the name "FL00375.DAT". You will need a routine to convert flight numbers to a 5 character string with leading zeros.

Later in this section you will see a package called the VIRTUAL_TERMINAL. The first version of this package body depended on the IMAGE attribute. To more the cursor to line seven column fifteen it was necessary to generate the string ASCII.ESC & "[7;15H". Figure 17 shows all the trouble I had to go

```
Figure 17. The IMAGE attribute is awkward to use.
---------------------------------------------------------

procedure Move_Cursor_To(LINE : Line_numbers;
                         COL  : Column_numbers) is
  L, C : string(1..3);
begin
  put(ESC & '[');
  if LINE < 10 then
    L(2..3) := integer'IMAGE(LINE);
    put(L(3..3));
  else
    L(1..3) := integer'IMAGE(LINE);
    put(L(2..3));
  end if;
  put(';');
  if COL < 10 then
    C(2..3) := integer'IMAGE(COL);
    put(C(3..3));
  else
    C(1..3) := integer'IMAGE(COL);
    put(C(2..3));
  end if;
  put('H');
end Move_Cursor_To;
```

through to get rid of leading spaces and adapt to the variable length strings produced by the IMAGE attribute.

These little quirks in the IMAGE attribute make it awkward to use in many applications, so I wrote the Image function in ASCII_UTILITIES. It does the same thing the IMAGE attribute does for integers, but it gives me the control I need over the length of the string and leading characters. Figure 18 shows how much easier it is to use the Image function than the IMAGE attribute.

Many people are surprised to discover the IMAGE attribute is not defined for the type float. That's because the attribute is defined only for discrete types, and type float is not a discrete type. I can understand why the LRM didn't require an IMAGE attribute for all data types, because the image of a composite data type can be complicated, especially if it is a record that has components that are composite data types. That doesn't mean it is impossible to define an IMAGE attribute for composite data types. In fact, Ada debuggers have to be able to display the contents of composite data types, and I have seen a couple that use aggregate notation to do that.

The lack of an IMAGE attribute for real data types is a nuisance if you want to output real numbers. You have no choice but to instantiate FLOAT_IO. Since I often don't use TEXT_IO, I didn't like that choice. Compilers are starting to get to the point where they can eliminate dead code, so linking all of TEXT_IO just to get a couple of string conversions may not be as bad as it used to be, but I still don't like to do it. The Fixed_Image and Float_Image functions in ASCII_UTILITIES give you better alternatives than instantiating FLOAT_IO.

```
Figure 18. The Image function is easy to use.
---------------------------------------------------------
  procedure Move_Cursor_To(LINE : Line_numbers;
                           COL  : Column_numbers) is
    L, C : string(1..3);
  begin
    put(ASCII.ESC & '[');
    put(string'(ASCII_UTILITIES.Image(LINE)));
    put(';');
    put(string'(ASCII_UTILITIES.Image(COL)));
    put('H');
  end Move_Cursor_To;
```

The Fixed_Image function will display a floating point number in fixed notation. You can select the number of characters before and after the decimal point. If you don't specify these values, the default is the minimum number of characters before the decimal point, and two characters after it.

The Float_Image function displays real numbers in exponential form. I thought about designing it so that it would allow you to specify the number of digits before the decimal point, but decided against it. When people output numbers in exponential form, they don't normally stick them in the middle of a line of text, and they don't normally want a variable number of digits before the decimal point. Numbers printed in exponential notation usually appear in columns on printouts, and I suspect people often just glance at the single digit in front of the decimal, and the exponent, to get an idea of the magnitude of the number. The numbers should line up neatly in the columns, so I precede the number with a leading space if it is positive. There isn't any way I can predict how many digits should be printed after the decimal point, so I made that a parameter. (I set the default to 5 digits to enforce my disposition toward 6 digit floating point numbers.)

I considered a single image function that would figure out if the user wanted fixed- or floating-point format, just as FLOAT_IO does. I decided not to for three reasons. First, combining the two routines results in a more complicated, and therefore less reliable, routine. Two simpler routines are less likely to contain an error, and are easier to test, than a single complicated one. Second, combining two similar but different routines sometimes forces some design trade- offs. For example, I wanted an AFT default of 2 for fixed format and 5 for exponential format. If I combined them I would have had to pick 2 or 5, or force the programmer to supply a value every time it is used. Third, the two names Fixed_Image and Float_Image make the program more readable. The reader doesn't have to know to look at the EXP field to see if it is zero or not, to tell him if the number will be printed in fixed- or floating-point format.

I considered making all the image functions in ASCII_UTILITIES generic, and decided not to. There isn't any need for generics. The Dimensionless functions can be used to convert dimensioned data types to Integer_32 or float and normal type conversions can be used to convert other special numeric data types to Integer_32 or float. Then the appropriate image function can be used on the equivalent Integer_32 or float value.

There isn't a predefined IMAGE attribute that will convert an integer from 0 to 9 (or 0 to F in hexadecimal) to the corresponding ASCII character. This would be a handy attribute to have if you needed to write a utility program that outputs files in octal or hexadecimal, or you wanted to print address locations in octal or hexadecimal. Several years ago I published a function in the Journal of Pascal, Ada & Modula-2 that fills this need. In those days I called it ASCII_Code_For, but to make it consistent with the other similar functions in ASCII_UTILITIES it is now called Image. This function doesn't let you specify the length of the return string because it doesn't return a string, it returns a single character. Instead it has a second parameter to let you specify the number base. This is necessary for it to check for digits out of range.

3.1.2. Qualified Expressions

There is one minor disadvantage to renaming the ASCII_Code_For routine to Image. It causes a compile-time error when you do something like this:

```
    TEXT_IO.put(Image(X,8));
```

Error messages are different for every compiler, so I can't tell you exactly what your compiler will say, but it will almost certainly contain the word "ambiguous." The problem is that TEXT_IO.put is overloaded for strings and characters. The ASCII_UTILITIES package contains two functions called Image, one of which returns a string and the other returns a character. Ada can't tell if you want to convert X to an eight character string and output the string, or convert X to an octal character and output the character. You have made an ambiguous statement and she refuses to try to guess what you mean.

There are three simple solutions. The first is to change the name of the character Image function back to ASCII_Code_For. I don't like that solution. (If I did, I wouldn't have changed it in the first place.) The second solution is to break the statement into two statements, making it clear what you want to do. For example, if you want to print an eight character string image, you could do the following:

```
S : string(1..8);
begin
     S := Image(X,8);
     TEXT_IO.put(S);
```

If you wanted to convert X to an octal digit you could do it this way:

```
C : character;
begin
     C := Image(X,8);
     TEXT_IO.put(C);
```

I prefer a third solution, which uses a qualified expression. The qualified expression is a rarely used Ada feature, but it comes in handy in situations like these. It is a way for you to tell Ada what you really mean. In this case it looks like this:

```
TEXT_IO.put(string'(Image(X,8)));
```

or

```
TEXT_IO.put(character'(Image(X,8)));
```

The function Image(X,8) can return either a string or a character. By enclosing it in parenthesis and qualifying it with a type, you tell Ada which one you want. Knowing the type of data returned by Image, she can select the correct put procedure.

3.1.3. VALUE Attribute

The VALUE attribute is the inverse of the IMAGE attribute, and it is also useful for I/O operations. For example, you can

```
get_line(TEXT,LENGTH);
X := integer'VALUE(TEXT(1..LENGTH));
```

to read an integer value from the terminal.

I have no complaint with Ada's standard VALUE attribute for discrete types, so I haven't written another version of it. The deficiency is that VALUE, like IMAGE is only defined for discrete types. Ada does not have a VALUE attribute for real numbers.

If she did, I probably wouldn't like it. Ada would probably be fussy about the format, just as she is for real literals in a program. That's fine for programmers. If a programmer wants to specify a real value he should be smart enough to know the correct syntax. You can't expect that kind of knowledge from a user. If you write software for military applications you have to remember that lack of a degree in computer science does not disqualify someone from joining the ranks of enlisted personnel. If you prompt a marine with "Enter the distance to the enemy position (in miles)", you better be prepared to accept "12" as an answer. If you insist on "1.20e01", the Marine is more likely to smash the weapon to bits than to figure out what you mean by "Data Entry Error!"

The Value function is complicated because it accepts any reasonable input and converts it to a floating point number. Real computer scientists will probably find this tolerance offensive, but that's the way it has to be in the battlefield. (If I were writing this function for use in a compiler, I would have required strict Ada syntax. You are permitted, even encouraged, to modify the Value function to make it less tolerant if that would make it more appropriate for your application.)

There is a Value function in ASCII_UTILITIES which converts a character in the range '0'..'9' or 'A'..'F' to a number from 0 to 15. It is the inverse of the single character Image function. It checks to make sure the input character is valid for the number base specified. The default is, of course, base 10.

You shouldn't need a qualified expression to distinguish the two value functions because both the input and output types are different. It is hard to imagine an ambiguous situation that might occur naturally, although I'm sure you could create one if you are devious enough.

3.1.4. Short Circuit Control Forms

I want to draw your attention to the short circuit control forms in Listing 16. Somewhere near line 40 you will find a section of code that looks like

```
-- reject special cases
if I < S_LAST then
     if S(I..I+1) = "-." or S(I..I+1) = "+." then
          case S(I+2) is
               when '0' .. '9' => null;
               when others     => raise CONSTRAINT_ERROR;
          end case;
     end if;
end if;
```

This section of code is designed to accept representations such as "-.7" and "+.05", but reject "-.e" and "+.". The variable S_LAST represents the last non-blank character in the string, and will be equal to S'LAST if there are no trailing blanks.

Suppose the string is "   +.". The pointer I has skipped over the leading blanks, and is pointing to the plus sign. It is true that S(I..I+1) = "+.". When the case structure tries to evaluate S(I+2), that is outside the range of S, a CONSTRAINT_ERROR will be raised. That's fine, because that's what I would do if S(I+2) existed and was not a digit. The CONSTRAINT_ERROR isn't a problem, it is a beneficial side effect that I have taken advantage of.

That isn't always the case. Consider this section of code just a little farther down the listing. I have replaced the short circuit control form (AND THEN) with the usual AND statement to illustrate a potential problem.

```
-- compute whole part
if S(I) = '.' then WHOLE := 0.0;
else
     FIRST := I;
     while I <= S_LAST and S(I) in '0'..'9' loop
               I := I+1;
     end loop;
     LAST := I-1;
     WHOLE := float(Integer_32'VALUE(S(FIRST..LAST)));
end if;
```

This part of the routine finds the first and last characters of the whole part of the number. That is, if the number is "-123.45", the routine will leave FIRST pointing to the number 1 and LAST pointing to the number 3, so S(FIRST..LAST) will be the string "123". This will work without error. Suppose, however, the string was "123". FIRST points to the 1, but CONSTRAINT_ERROR might be raised looking for the 3.

The problem is in the statement while I <= S_LAST and S(I) in '0'..'9' loop when I = S_LAST+1. If the program tries to evaluate S(S_LAST+1) to see if it is in '0'..'9', CONSTRAINT_ERROR will be raised. You can't be sure the program will check I <= S_LAST and realize the whole expression must be false before finding the character in S(S_LAST+1) because section 4.5 paragraph 5 of the LRM says that they are "evaluated in some order that is not defined by the language."

The solution is to use the short circuit control form. The statement while I <= S_LAST and then S(I) in '0'..'9' loop tells the computer to evaluate I <= S_LAST first, AND THEN evaluate S(I) only if the first part is true. You will find several examples of the short circuit control form AND THEN in the ASCII_UTILITIES.Value function.

3.1.5. Character Conversions

If you ask a user to input a file name, you may need to convert it to all upper-case (or all lower-case) letters. You might want to write a spelling checker which finds words in a dictionary file, and you don't care if the words are capitalized or not. You may want to remove the underscores from the image of an enumeration type before printing it. The ASCII_UTILITIES package contains routines called Upper_Case, Lower_Case, and Change that will do this for you.

These text processing functions can be used in combination to make an enumeration output look prettier. For example, a poker program may have an enumeration type with the value THREE_OF_A_KIND that must be displayed. The IMAGE attribute first converts it to the string "THREE_OF_A_KIND". The Lower_Case string function can then convert all the letters to lower case, and the Change string function can replace the underlines with blank spaces. This produces "three of a kind". If you like, you can use the Upper_Case character function to capitalize the first character to get "Three of a kind".

3.1.6. Ada Strings

Ada's fixed strings are hard for some people to get used to. If you have done much programming in other languages, you probably think dynamic (variable length) strings are essential. I thought so. That's why I published an improved version of Sylvan Rubin's dynamic string package ([4])([5]). But the more I used dynamic strings, the more I realized they were a bad idea. I discovered that, sooner or later, you had to decide how many characters the string could hold. Discovering it later usually caused trouble. I abandoned the use of dynamic strings, and now exclusively use fixed strings.

---

[4] Dr. Sylvan Rubin, Dynamic String Functions in Ada, Journal of Pascal, Ada & Modula-2, Vol. 3, No. 6, Nov./Dec.1984.

[5] Do-While Jones, Ada Dynamic Strings Revisited Part 2, Journal of Pascal, Ada & Modula-2, Vol. 5, No. 3, May/June 1986.

When using fixed strings, is is occasionally necessary to add spaces to the end of a short string to fill out a variable that was declared as a longer string. It might also be necessary to truncate a long string to make it fit in a shorter string. The String_Copy procedure was designed to do this.

String_Copy can be useful when dealing with enumeration images. For example, if type Colors is (RED, GREEN, BLUE); and you use Colors'IMAGE to convert a value to a character string, the result may be 3, 4 or 5 characters. If you want the result to always be five characters, you can do the following:

type Colors is (RED, GREEN, BLUE);

```
          X : Colors;
          S : string(1..5);
     begin
          String_Copy(FROM => Colors'IMAGE(X), TO   => S);
```

```
Figure 19. ASCII_UTILITIES demo.
------------------------------------------------------

with ASCII_UTILITIES, TEXT_IO;
procedure ASCII_UTILITIES_Demo is

  type Values is (NOTHING, A_PAIR, TWO_PAIR,
    THREE_OF_A_KIND, A_STRAIGHT, A_FLUSH,
    A_FULL_HOUSE, FOUR_OF_A_KIND,
    A_STRAIGHT_FLUSH, A_ROYAL_FLUSH);

  S : string(1..16);

begin
  TEXT_IO.put_line("Possible poker hands are:");
  for i in Values loop
    -- Make all values 16 character strings,
    -- padding with nulls if necessary.
    ASCII_UTILITIES.String_Copy
      (FROM => Values'IMAGE(i),
       TO   => S,
       FILL => ASCII.NUL);
    -- Convert all character to lower case.
    S := ASCII_UTILITIES.Lower_Case(S);
    -- Change underlines to spaces.
    S := ASCII_UTILITIES.Change(S);
    -- Capitalize the first letter.
    S(1) := ASCII_UTILITIES.Upper_Case(S(1));
    -- indent and print it
    TEXT_IO.put_line("   " & S & ',');
  end loop;
end ASCII_UTILITIES_Demo;

--------------------------------

When directed to a printer, the output looks
like this:

Possible poker hands are:
   Nothing,
   A pair,
   Two pair,
   Three of a kind,
   A straight,
   A flush,
   A full house,
   Four of a kind,
   A straight flush,
   A royal flush,
```

Consider the poker example again. The value of the hand might be NOTHING, A_PAIR, TWO_PAIR, THREE_OF_A_KIND, A_STRAIGHT, A_FLUSH, A_FULL_HOUSE, FOUR_OF_A_KIND, A_STRAIGHT_FLUSH, or A_ROYAL_FLUSH. The length of image of the value could be six to fifteen characters long. Figure 19 shows how the text processing functions can format the output for display. (Note that Figure 19 shows what you will get if you direct the output to the printer, and what you should get if you direct the output to a CRT screen. I discovered that something in MS-DOS or my EGA card substitutes blanks for nulls, because all the commas line up vertically if I look at the output on the CRT.)

3.2. MONEY_UTILITIES package

Money is an important part of our daily life, and naturally it becomes a subject of a variety of accounting programs. There are programs that compute the mortgage payments, tax liabilities, and interest on savings, to name a few. The MONEY package (Listings 17 and 18) is useful for Ada programs that do financial calculations in American dollars. It is included in this section because it contains Image and Value functions, needed for input and output of monetary figures.

Notice that the MONEY package builds on the concepts introduced in the previous section. We don't want to multiply dollars times dollars to get dollars squared, and if we did we certainly wouldn't want to assign it to an object of type dollars. Therefore we should use a dimensioned data type to represent monetary values. The question is, which one?

Fixed-point is a logical choice to represent dollars, but I explained why that isn't a good idea when I told why I never wrote a FIXED_UNITS package. A floating-point data type could be used to represent dollars. If we do that, the accuracy will vary with the size of the amount because there will be a fixed number of bits to represent the whole number of dollars plus the fractional dollars (cents). Therefore the dollar figures would be accurate to a certain percentage, but not to a certain number of cents. The accuracy is much worse than expected if calculations compute a small difference between large sums of money. Floating- point isn't a good choice for representing money.

Practical arguments aside, the real reason you shouldn't use a fixed- or floating-point data type to represent money is that money is a discrete quantity rather than a continuous one. If money was specified in terms of ounces of gold instead of individual pennies, then a real variable would appropriate for money. When working with money we are really dealing with a finite number of individual, countable things, so an integer representation is the philosophically correct one.

If we decide to use an integer data type to represent cents, then we must decide how many bits to use. Sixteen bits would allow us to represent values from -$327.68 to $327.67. That's far too small. Thirty-two bits range from -$21,474,836.48 to $21,474,836.47. While this isn't sufficient range to represent national budgets, it is certainly large enough to figure the mortgage on any house I am likely to buy.

Having made this decision, the MONEY package declares a data type Cents which is a dimensioned 32-bit integer. It also provides Image and Value functions to convert between strings and Cents. Notice that even if I had written the ASCII_UTILITIES.Image and ASCII_UTILITIES.Value as generic functions, I could not have instantiated them for type Cents because I want to be able to use a decimal point, a dollar sign, and commas in the string.

The Width function tells the size string required to represent the value. This is essential for declaring strings of the proper size.

Figure 20 is a simple program that demonstrates how to use the money package. It prompts the user to enter the price of an item, the sales tax rate, and prints the total cost to the customer. This isn't a terribly useful program, but it does show how to declare objects of type Cents, and shows how to input and output them.

```
Figure 20. Sales Tax.
----------------------------------------------------------
-- This program shows how to use the MONEY package.

with MONEY, ASCII_UTILITIES, STANDARD_INTEGERS;
with TEXT_IO; use TEXT_IO;
procedure Sales_Tax is

  PRICE, TAX, COST : MONEY.Cents;
  RATE             : float;
  TEXT             : string(1..79);
  LENGTH           : natural;

  function "*"(LEFT : MONEY.Cents; RIGHT : float)
      return MONEY.Cents is
    use STANDARD_INTEGERS;
    X      : float;
    RESULT : Integer_32;
  begin
    -- Compute exact amount
    X := float(MONEY.Dimensionless(LEFT)) * RIGHT;
    -- Round to the nearest cent
    RESULT := Integer_32(X);
    -- Convert the answer to Cents
    return MONEY.Type_Convert(RESULT);
  end "*";

  use MONEY; -- for "+" operator in COST := PRICE + TAX;

begin
  put("What is the cost of the item? ");
  get_line(TEXT,LENGTH); new_line;
  PRICE := MONEY.Value(TEXT(1..LENGTH));
  put("What is the sales tax rate? " );
  get_line(TEXT,LENGTH); new_line;
  RATE := ASCII_UTILITIES.Value(TEXT(1..LENGTH));
  -- if RATE > 1, the user must have entered
  -- a percentage (i.e. 6%) instead of 0.06.
  if RATE > 1.0 then RATE := RATE / 100.0; end if;
  TAX := PRICE * RATE;
  declare
    TAX_STRING : string(1..MONEY.Width(TAX));
  begin
    TAX_STRING := MONEY.Image(TAX);
    put_line("The tax on that item is "
             & TAX_STRING);
  end;
  COST := PRICE + TAX;
  put("Your total cost is ");
  put(MONEY.Image(COST));
  put_line(".");
end Sales_Tax;
```

Cents is derived from Units in INTEGER_UNITS. I can multiply Cents by dimensionless integers, but I can't multiply them by dimensionless real types because there isn't a multiply operator for Units times float in INTEGER_UNITS. That wasn't an oversight. I once had a floating-point multiply in INTEGER_UNITS, but I decided to take it out. I wanted to force myself to think about how to handle the fractional results that are likely to occur.

When PRICE is multiplied by RATE, the resulting TAX may include fractional pennies. In California, sales tax is rounded to the nearest cent. If I lived in a state that insisted on rounding any fraction of a cent of sales tax to the next higher cent, I could have written that into the multiplication algorithm shown in Figure 20.

The body of Figure 20 shows how I used the Value function in MONEY to convert the user's entry from a character string to the PRICE in Cents. If you compile and run this example you might have some fun trying experiments. You will find that you can enter a price of $5.00, $5, 5, 5.0, $05.00, or several other ways you might imaging. The MONEY.Value function lets you embed dollar signs and commas in the price. If you try to enter a a tax rate with a dollar sign in it, you will get a CONSTRAINT_ERROR. That's because the rate is a pure number, interpreted by ASCII_UTILITIES.Value, not MONEY.Value.

I didn't want to make this example too complicated and distract from the use of the MONEY package, so I didn't check for errors (like negative tax rates). If you want to fool around with this example, you might try detecting the presence of a percent sign in the TEXT string, and dividing the RATE by 100 only if the user entered a percent sign.

I put a block structure in the middle of the Sales_Tax program just to show you how you could use MONEY.Width to declare a string like TAX_STRING. I don't normally do that. I usually create the strings on the fly, like I did when I printed COST. (There I didn't bother to create COST_STRING. I just put the MONEY.Image function inside the put procedure.)

I used TEXT_IO as a user interface in this example. I followed every get_line with a new_line. That may or may not cause a blank line after each input, depending on your operating system. If you leave the new_line calls out, you may (or may not) have superimposed input prompts. This is just one of many problem you will discover if you try to use TEXT_IO as a user interface. Fortunately, you don't have to use TEXT_IO if you don't want to. That's our next topic.

3.3. TEXT_IO package

When most Ada programmers think of IO, the first thing that comes to mind is TEXT_IO. Although Ada doesn't have any built-in IO instructions, the language designers realized that Ada would be used for more applications than just embedded computers. They knew many general purpose applications would require interfaces to common peripherals (disks and terminals), so they filled Chapter 14 of the LRM with some standard IO packages. These packages, such as TEXT_IO, are provided for your convenience. You may use them if you like, but you don't have to.

I often use TEXT_IO in the code examples I publish. This should not be considered an endorsement of TEXT_IO. The fact is, I don't like TEXT_IO very much. I use it because all Ada programmers are familiar with it. If I used SCROLL_TERMINAL or FORM_TERMINAL, the unusual user interface would distract readers from the point of the example.

3.3.1. What's Wrong With TEXT_IO

I don't really have many complaints with TEXT_IO as an interface to text files, but it makes a terrible user interface. TEXT_IO treats the user's terminal just like a file, and that creates input problems. Files never make mistakes, so they don't need a rub out key. Files never enter passwords that they don't want echoed to the screen. Files never want to insert or delete text. Files never want to clear a screen or move a cursor. Files never realize the program has run amok and try to send an unsolicited CTRL-C to stop the process. Users often want to do all of these things, but TEXT_IO won't let them because it wasn't designed to support users.

The second problem with TEXT_IO is that it is heavily dependent upon the host operating system. That means it works differently on different computers, which makes it difficult to move programs from one computer to another. For example, a program containing the line TEXT_IO.put_line(OUTPUT_FILE,"Some Text"); may write "<LF>Some Text<CR>" or "Some Text<CR><LF>", depending on how the operating system service writes lines of text. This causes minor, but annoying, problems when you write a file on one computer and try to read it on another.

You get some nastier surprises when you instantiate INTEGER_IO for integers and then call get(X); to read the integer X. Suppose while entering X the user types the wrong character and uses the rub out key to delete it. Some operating system calls will respond to the editing command and pass the corrected number X to INTEGER_IO for processing. Other operating system calls will just leave the rub out character in the middle of the string of digits. INTEGER_IO will recognize that rub out is not a decimal digit and raise DATA_ERROR, which at best will prompt the user to enter the number again.

Even if the user doesn't make a typing error, there are still problems. INTEGER_IO.get(X) will read all the characters up to, but not including, the end of line marker. These characters will be correctly converted to the number X, but the input buffer pointer will be sitting just in front of the end of the line marker. If you then call get_line(TEXT,LENGTH), it will return a null string. You have to remember to write get(X); skip_line; every time you get an integer.

Since input data editing may or may not be done by the operating system called by the get procedure, you never can tell if CTRL-X will erase a whole line, or if backspace will be the same as delete. These things all combine to frustrate the user and make Ada appear to be a user-hostile language.

I got fed up with TEXT_IO in a hurry, and wrote a set of replacement user interfaces. These packages are called VIRTUAL_TERMINAL, SCROLL_TERMINAL, and FORM_TERMINAL. They don't have a lot of state-of-the-art bells and whistles (like graphics and pop-up windows) because they have to run on "glass teletype" terminals, but they do provide a portable, consistent, friendly user interface.

3.4. VIRTUAL_TERMINAL package

Terminals are notoriously inconsistent when it comes to control codes. They all have different control sequences for clearing the screen and moving the cursor. The VIRTUAL_TERMINAL hides all these differences. The package specification is shown in Listing 19.

3.4.1. Guaranteed Functionality

As you can see, it supports a screen with 79 columns and 23 lines. Everybody knows terminals always have at least 80 columns, but the terminal's response after the 80th column has been printed is uncertain. Some terminals stay at column 80 and overprint the last character entered. Other terminals automatically generate a carriage-return/line-feed sequence and go to the next line. Often the action after the 80th character is programmable. On the other hand, all terminals treat columns 1-79 the same. I was willing to sacrifice 1 column to avoid portability problems. That's also the reason why I used a conservative 23 lines, even though common American terminals have 24 or 25 lines. A 23 line, 79 column screen is the most guaranteed functionality I can expect from every CRT.

The VIRTUAL_TERMINAL has four cursor control (arrow) keys, 20 function keys, and the INSERT, DELETE, TAB and BACK_TAB keys. Not all terminals have all these keys, so provisions have been made for control characters to simulate these 28 special keys. I'm aware that many terminals have more special keys than these, but I'm designing for maximum portability, not maximum performance. Even with this limited capability and designing the package with portability in mind, there are problems porting the VIRTUAL_TERMINAL to DEC terminals on VAX/VMS. (These problems are described in a later section.)

The 20 function keys are converted to a two-character control string. The string begins with CONTROL-F and is followed by a character '1' through '9' or 'A' through 'K'. The remaining eight keys are converted to a single control character. (Refer to the get procedure specification in Listing 19 to see what they are.) Therefore, those 8 keys can be simulated by entering the control code. For example, if a keyboard lacks a DOWN arrow key, the user can use CONTROL-D instead.

3.4.2. Information Hiding

The VIRTUAL_TERMINAL specification give you a procedure that clears the screen, as well as procedures that position the cursor. These procedures hide the control sequences used by the physical terminal from the application program. This information hiding is the key to portability.

The VIRTUAL_TERMINAL needs three services from the underlying operating system. (1) Send a single character to the screen, (2) get a character from the keyboard without echoing it, and (3) find out if there are any unprocessed keystrokes. Every operating system does this differently, and the VIRTUAL_TERMINAL body hides these differences from any application program that uses it.

3.4.2.1. Alsys VIRTUAL_TERMINAL body.

If you are using the Alsys compiler on an IBM PC AT compatible system, these three services are all provided by the Alsys DOS package. The VIRTUAL_TERMINAL package body that works for Alsys programs is shown in Listing 20. The procedure DOS.Display_Char sends a character to the display. The procedures for clearing the screen and moving the cursor can use DOS.Display_Char because it will pass all ASCII characters (even ESC) to the display. (I point this out here because it is different from the Meridian implementation of the DOS interface.)

The function DOS.Read_Kbd_Direct_No_Echo gets characters from the keyboard directly (that is, without interpreting control sequences) and without echoing them to the screen. The get procedure uses this function to fetch keystrokes. It converts the special keys to the standard control keys before passing them to the application program.

The Keyboard_Data_Available function simply renames DOS.Kbd_Data_Available. Notice I didn't use the rename statement because I would have had to have put that in the package specification. I wanted to keep the package specification the same for all implementations, and just change the body. I hope that the global optimizer will remove the double call for me. (It doesn't really matter if it does or not, because human reaction time is much slower than the few wasted microseconds in the double function call.)

The control procedures use the put procedure to send terminal-specific control codes. For example, Clear_Screen uses put to send the control string ASCII.ESC & "[2J" to the screen. The cursor control procedures build similar strings and use put to send them to the display. The exact form of the control string depends on the hardware you are using, so you will probably have to change the escape sequences if you are using a different terminal. That's why you need different bodies for different systems.

The Move_Cursor_To procedure is interesting. It needs to construct the control string ASCII.ESC & '[' & L & ';' & C & 'H' where L is the string representation of the LINE number and C is a string representing the COL. There can be no embedded blanks. You've already seen Figure 17, which shows all the trouble I had to go through to move the cursor before I had the ASCII_UTILITIES.Image function.

The get procedure is a little more complicated than you might expect. It clearly does a lot more than just get the keystroke. It has to check to see if the keystroke is one of the special keys, and if it is then it converts it to the standard control code. On the IBM PC AT the special keys are sent as a two character code. The first code is always 00 and the second code is a unique number. For example, the LEFT arrow key is 00 75. The RIGHT arrow key is 00 77. The get procedure recognizes the 00 as a special-key flag and then reads the second number to find out which key it is.

If the special key is a function key in the range 1..20, the get procedure returns CONTROL_F and stores the number 1..20 in a hidden variable F_KEY. The application program can get this number by calling the function Function_Key. It is possible that the terminal doesn't have function keys, and the user simulated the function key by pressing CONTROL_F followed by a letter or digit. Since the computer is probably faster than the user, there is a danger that the application program will recognize the CONTROL_F input and call Function_Key before the user has a chance to press the letter or digit. That's why F_KEY is reset to 0 each time it is read. If the Function_Key function finds that F_KEY is 0 it knows the user hasn't pressed the letter or digit yet, and waits to read the next keystroke itself.

3.4.2.2. Meridian VIRTUAL_TERMINAL body.

The Meridian version of the package body is shown in Listing 21. It shows how different two implementations on the same machine can be. Instead of a general purpose package like the Alsys DOS package, Meridian sells several special purpose utility packages. If you know a lot about MS-DOS, you can use the INTERRUPT package to access MS-DOS directly. That may be too complicated for some programmers, so Meridian has some less flexible, but simpler, user interface packages. Two of these packages are TTY and CURSOR.

The TTY package calls put to send a single character to the display. It seems to be just like the Alsys DOS.Display_Char, but it isn't. If you try to use TTY.put to send ASCII.ESC & "[2J" to the display to clear the screen, it won't work. It filters out the ESC and just writes [2J on the screen. Similarly, ASCII.ESC & "[C" won't move the cursor right one space. That's why I had to use special routines like TTY.Clear_Screen and CURSOR.Right in the Meridian body.

The TTY.get procedure is similar to the Alsys DOS.Read_Keyboard_Direct_No_Echo function. The TTY.get procedure has two boolean parameters, DIRECT and NO_ECHO, which both must be set to TRUE to achieve the desired effect.

Finally, the TTY.Char_Ready function is effectively renamed to Keyboard_Data_Available by enclosing it in a function body.

If you look at the Move_Cursor_To procedure, and compare it with the Alsys version, you will see an offset has been subtracted. That's because Alsys counts rows and columns starting with 1, and Meridian starts counting at 0. If you carefully compare the two bodies you will find more small differences. The important point to make is that all of these differences would appear in every application program if they weren't carefully confined to the VIRTUAL_TERMINAL body.

3.4.3. Visual Attributes

Terminals usually have different visual attributes. That is, the characters can be bright or dim, blinking or steady, normal or reverse video. An interesting exercise is to make a copy of VIRTUAL_TERMINAL Version 1 and rename the copy Version 2. Then modify Version 2 to include attribute setting procedures called Use_Bright, Use_Dim, Use_Blinking, and so on. You may want to add boolean functions Is_Bright, Is_Dim, and so on, that tell the current status of the visual attributes.

3.4.4. VIRTUAL_TERMINAL Uses

The VIRTUAL_TERMINAL can be used for screen-oriented displays. It is handy whenever you want to move the cursor all over the screen and write text fragments in different places. You'll see an example of this in the FORM_TERMINAL.Create procedure later in this section.

Although the VIRTUAL_TERMINAL can be used as an end product, that isn't really its main use. The VIRTUAL_TERMINAL is most valuable as a portable foundation that can be used as the base for a more useful terminal package. You are about to see two such packages, SCROLL_TERMINAL and FORM_TERMINAL. These two packages are built on top of VIRTUAL_TERMINAL, so it isn't necessary to have different bodies for every implementation. If you can port the VIRTUAL_TERMINAL to work with a different physical terminal or different operating system, then you have ported SCROLL_TERMINAL and FORM_TERMINAL as well.

3.5. SCROLL_TERMINAL package

The SCROLL_TERMINAL is a portable, general purpose interface. It is called "scroll" because new data is printed at the bottom of the screen, causing old data to scroll off the top.

3.5.1. Reusability and Consistency

The user interface is often one of the most complicated parts of an application program. That makes it an important candidate for reuse. After all, isn't it better to reuse the components that are hardest to write, than to reuse trivial ones?

Auto manufacturers recognize that people are creatures of habit, so they put controls where people expect them to be. Maybe the steering column isn't the best place for the turn signal, but any auto maker who moved it to a "better" place would have trouble selling cars. Unfortunately, many programmers don't have the good sense auto makers do. Every time they write a program they redesign the user interface. Sometimes they use TAB to advance to the next item on a menu, other times they use the RIGHT arrow key to do that. Sometimes backspace acts as a delete, other times it doesn't. It can drive a user nuts.

Windows are becoming popular and are starting to put an end to unique interfaces. When application programs use commercially available window products they not only benefit by reusing tested user interfaces, they also present a familiar interface to the user. A complete window package with graphics and a mouse interface is too complicated to be included in a book of intermediate Ada examples, but SCROLL_TERMINAL gives you the same benefits of reuse and consistency. It is a simple, standard user interface which can be used by a variety of application programs.

3.5.2. Layering

Often it is a good idea to build device handlers layer upon layer. That's what I've done here. The SCROLL_TERMINAL (Listing 22) is built on the VIRTUAL_TERMINAL, which hides hardware specific differences. There is no need for special SCROLL_TERMINAL bodies for each of the different implementations. The SCROLL_TERMINAL has the same number of lines and columns as the VIRTUAL_TERMINAL, and propagates the VIRTUAL_TERMINAL.PANIC exception by renaming it. Therefore, the application program doesn't need to know that there is a VIRTUAL_TERMINAL under the SCROLL_TERMINAL.

3.5.3. SCROLL_TERMINAL Features

The SCROLL_TERMINAL boasts features you won't find in TEXT_IO. It lets you check the keyboard to see if the user has entered any keystrokes, and you can flush the typeahead buffer to discard any entries that may have been typed (but not processed) before the prompt was displayed. You can turn the character echo on or off. There are several new get procedures which return strings of the proper length (padding with blanks if necessary). The new input routines include a prompt, like a BASIC input statement. (If you don't want a prompt, then use the null string as a prompt.)

The SCROLL_TERMINAL has an interesting way of handling default responses. It works like this: The screen shows the prompt and the default response. If you press RETURN, it takes the default. If you begin typing a new response, the default is automatically erased. If the default is almost exactly what you want, you can edit the default response.

The LEFT and RIGHT arrow keys can be used to move the cursor without changing the characters underneath the cursor. If the keyboard doesn't have arrow keys, CONTROL-L and CONTROL-R can be used as LEFT and RIGHT arrow keys. BACKSPACE (CONTROL-H) deletes the character to the left of the cursor and moves the cursor back one space. DELETE (CONTROL-E) erases the character covered by the cursor. INSERT (CONTROL-A) adds characters at the cursor location without destroying any existing text.

Whenever you press RETURN, the entire response showing on the screen is taken, regardless of the cursor position. (You don't just get the characters to the left of the cursor.) This was done at the request of a customer who insisted on a constant policy of "What you see is what you get."

The SCROLL_TERMINAL also generates an exception called NEEDS_HELP whenever the user presses the question mark key. This is an important feature you will see demonstrated often in the later programming examples.

3.5.4. Compatibility

Even though I don't like TEXT_IO, I know there are a lot of application programs that have already been written using TEXT_IO as the user interface. Therefore, I should make it as easy as possible for those programs to use SCROLL_TERMINAL instead of TEXT_IO. The routines with TEXT_IO names (put, put_line, new_page, get_line, and so on) are identical to TEXT_IO so existing programs that use TEXT_IO for a user interface can use SCROLL_TERMINAL instead simply by substituting with SCROLL_TERMINAL; use SCROLL_Terminal for with TEXT_IO; use TEXT_IO;.

3.5.5. Hiding Details in the Package Body

The package body is shown in Listing 23. It makes a distinction between cursor positions and column numbers. The column can be 1-79 but the cursor can be at position 1-80. This was done so the cursor will always be to the right of the character just entered. The application program doesn't need to know about Cursor_positions, so this data type is hidden in the package body.

3.5.6. Coupling

If one module affects another module, those two modules are said to be coupled. If modules are too tightly coupled, then there are major maintenance problems. When you change one module it forces you to change another, which is coupled to another so that other module must also be changed, and the ramification of a change ripple through the whole system. Too much coupling is bad.

If you take the extreme position that all coupling is bad, then you can never build a working system because all the modules in a system have to work together to achieve the goal. They can't do that if they are completely independent.

The coupling quandary is like networking computers. Many people want their computers to be networked (coupled) together so they can share data, but as soon as they do that they run risks. A network failure could keep them from accessing vital data. A hacker could break into one part of the network and gain access to the whole network. When networking computers you want to control the coupling, so authorized users can safely pass data (even if part of the network fails), but unauthorized users can't get at any of the data. The same is true of coupling software modules. You need controlled coupling.

One of the best ways to control coupling between modules is to pass all information from one to the other using parameter lists. This method allows Ada to check for consistency between modules, and helps programmers see how modules are coupled. I pass information between modules using parameter lists whenever I can, but there are some times when this isn't practical. The SCROLL_TERMINAL body is a good example.

There are three variables in the package body that are used to couple several routines together. COLUMN_NUMBER tells where the cursor is. TAB_STOPS remembers how many columns there are between tab stops. ECHO is used to decide if input characters should be echoed or not.

Let's look at ECHO first. All of the input procedures need to know if they should echo characters to the screen as they are typed. We could do that by adding one more parameter to all of the input procedures. This boolean parameter would tell the input routine if it should echo or not. You could put this parameter last in the list, and give it a default value of TRUE, and it wouldn't be too awkward, but you would force every application program that ever needs to control the echo to keep track of the echo status itself.

I elected to add Echo_On and Echo_Off procedures to the SCROLL_TERMINAL body to make it easier on the application program. Since the normal mode is Echo_On, the elaboration of SCROLL_TERMINAL always calls Echo_On. The application program can call Echo_On or Echo_Off whenever it wants, and does not need to remember the current echo status.

The method works by coupling the Echo_On and Echo_Off procedures to all the get procedures, using the shared variable ECHO. In so doing, I intentionally violated a software quality assurance guideline that says, "There shall be no hidden couples between modules." If management got really nasty about it, I could get rid of the ECHO variable and clutter all the input routines with boolean parameters, without doing too much damage to the overall design.

The situation isn't so simple with COLUMN_NUMBER and TAB_STOPS. These variables provide a straight-forward (hidden) way of coupling the output procedures. Look at what has to happen.

The procedure put has to do more than just output characters. It also has to keep track of the cursor position. It has to do this so it can expand TAB characters. If the character to be printed is a printable character, it generally prints the character and moves the cursor to the next location. The one exception is when the cursor is at column 80. Then the new_line procedure is called and the character is printed in the first column of the next line, and the cursor moves to column 2.

Non-printable characters are handled specially. CARRIAGE_RETURN sets the cursor and the COLUMN_NUMBER back to column 1. LINE_FEED and BELL are sent to the display without affecting the COLUMN_NUMBER. TAB is sent as one or more spaces (until COLUMN_NUMBER is a multiple of TAB_STOPS). Other non-printable characters (ESCAPE, for example) are replace by BELL characters to prevent programmers from trying to send terminal-specific control sequences to the screen. (The SCROLL_TERMINAL is a portable package. If it lets terminal-specific codes through, then you can't be sure an application program can be ported to another system.) The procedure Set_Col acts like a TAB, except it spaces over to the specified column regardless of TAB_STOPS.

So Set_Tab, Set_Col, new_line, and put are all coupled through COLUMN_NUMBER and/or TAB_STOPS. I'm a really clever fellow, so I probably could figure out a way to pass COLUMN_NUMBER and TAB_STOPS all over the place using parameter lists, but what a maintenance nightmare that would be!

The guideline is correct. You should generally avoid using shared variables as hidden couples between modules. The guideline shouldn't be considered to be absolute. There are times when the guideline should be violated. Some would argue that the SCROLL_TERMINAL body is a module, and that Set_Tab, Set_Col, new_line, and put are cohesive parts of one module, so the guideline hasn't been violated. That's just avoiding the issue. You have to recognize that there are rare instances when it is better to use hidden coupling than visible coupling.

3.5.7. Module Partitioning

The SCROLL_TERMINAL package body is broken into two files (Listings 23 and 24). The first file contains the main part of the package body, and the second file is the Get_Response subunit. The package body is five pages long, and the subunit is five pages long. If I left Get_Response in the package body, then the body would have been ten pages long, and that is too long.

Some people would say that even five pages is too long, and I generally agree with them. I try to keep each compilation unit to three pages or less, but I don't have a fixed rule, "Thou shalt not write any compilation unit longer than three pages." Guidelines have to be tempered by common sense.

I could have reduced the size of the SCROLL_TERMINAL body by turning put_line, New_Page, Set_Col, Set_Tabs, and several other small procedures into separate units. That might have made the body small enough to satisfy someone's software quality assurance guideline, but would that really be an improvement? Does it really help to break SCROLL_TERMINAL up into a dozen files, many of which contain procedures with only six statements? The file headers would take up more space than the executable code! (That's not a disk space problem, it is a visual clutter problem. The code gets lost in the boilerplate "information" people unconsciously skip over.)

I'm not arguing in favor of big compilation units. There are good reasons for keeping compilation units small. When compilation units get large, it is hard to find a particular piece of code. Often a module is too large because it is trying to do too many things at once, and should be broken down into several smaller modules that each do a single thing. Whenever you have a module over several pages long, you should seriously consider dividing it into smaller pieces.

Partitioning the code into smaller modules is a good idea most of the time, but there comes a point where partitioning hurts more than helps. I have friends working on a project where management insists that every subprogram must be a separate module. If I wrote SCROLL_TERMINAL for that customer I would have to treat Set_Tabs as a configuration controlled module, complete with pseudocode description, structure chart, data dictionary, formal walkthrough, requirements traceability, module test plan, and integration test plan. (My friends' project is currently one year behind schedule, about to announce another year schedule slip, is way over budget, with no end in sight. I think I know why.)

3.5.8. Limited Name Space

Another factor you need to consider when partitioning a program into modules is that the more modules you have, the more module names you need to make up. You may think I'm joking, but when you have a big program with several hundred modules, it gets tough to find a name that is short, meaningful, and hasn't already been used. This is a serious problem in Ada because subunit names can't be overloaded.

The SCROLL_TERMINAL body has two overloaded subprograms called put. One put takes a character as a parameter, the other takes a string. Both exist blissfully in the package body. I can make either one of them a subunit, but if I try to make both of them separately compilable subunits, I can't do it. I would have to rename one, hide one in yet another package, or think of something else equally clumsy. Sometimes it may be better to let a module get a little bigger than the guidelines suggest, rather than be forced to do something creative to a procedure name to resolve a name clash so you can use separate subunits.

The SCROLL_TERMINAL body and Get_Response subunit are long, but I don't think they would be improved any by more partitioning. Partitioning would just make configuration management more difficult because you would have so many more files with strangely named modules in them.

3.5.9. The Top is at the Bottom

Ada makes it easy to write programs from the top down, but when you are finished writing it you will find the top level logic is at the bottom of the listing. You can see this when you look at the Get_Response subunit.

Ada needs to have all the minute details described to her before she is willing to look at the big picture. In the Get_Response subunit she needs to be told all about Beep, Forward, Backup, and so on, before she will even consider the main sequence of statements. My human mind balks at this. I can't help thinking, "Why are you telling me about this Beep? What has Forward got to do with getting a response from the user?" I find it much easier to understand a listing by going to the end and working backwards.

With this in mind, lets skip all the way down to the main begin statement in Listing 24. The program begins by setting the INSERT_MODE to FALSE. This means characters entered will type over existing ones, rather than shoving existing characters to the right to make room for a new character to be inserted at the cursor location. Then the DEFAULT response is put in a temporary BUFFER, and the BUFFER is displayed if the ECHO is enabled. If ECHO is disabled, then blanks are displayed instead. Writing the contents of the BUFFER (or blank spaces) to the screen moves the cursor to the end of the BUFFER. X is a scratch variable used to keep track of where the cursor is, so it is set to the end of the BUFFER. Then the Backup procedure is called as often as necessary to move the cursor back to the beginning of the BUFFER. Backup automatically adjusts X and the COLUMN_NUMBER.

All of these actions happen in the twinkling of an eye. You probably won't see them unless you look closely at the screen, or are using a 1200 baud modem. To the casual observer, the default appears and the cursor is sitting at the beginning of the default. The program is now sitting at the VIRTUAL_TERMINAL.get(C); line, waiting for the user to enter a character.

The first character the user enters may be special. If the user presses INSERT (CONTROL_A), the INSERT_MODE is set to TRUE. This allows subsequent characters to be added to the beginning of the default. If the first character entered is DELETE (CONTROL_E), the first character of the default is erased but the remainder of the default remains. If the first character is a RIGHT arrow (CONTROL_R), the cursor moves one space to the right, without affecting the default characters under the cursor. If the first character is the RETURN key, the DEFAULT response (visible or not) is returned as the user's input and the subunit is done.

Often the first character will be none of the above. In that case, the default response is erased and the first character is processed as a normal keystroke input. The default is erased by filling the BUFFER with blanks and writing the BUFFER to the screen. This puts the cursor at the end of the BUFFER again, so Backup needs to be called to move the cursor back to the beginning of the BUFFER. Since no characters have been processed yet, SIZE (the number of valid characters in the buffer) is set to 0.

After the initial character is entered, the program goes into a loop that gets a character and processes it. The Process_Character procedure puts the keystrokes in the BUFFER and sets DONE to TRUE when the user presses RETURN. The DONE flag is used to exit the loop. When this happens the BUFFER and the SIZE are passed back to the main program and the cursor is moved to the beginning of the next line.

That is the top level description of the logic flow. It gives a general description of what Get_Response does. Suppose we want more detail about the Process_Character routine. We go to the next lower level by backing up in the listing.

The Process_Character procedure is found just before the main begin. Process_Character is always called after the user has entered the character C. A case structure decides what to do with C. For example, if the

C was the RETURN key, then all that needs to be done is to set DONE to TRUE. If it was a BACKSPACE character, then Rubout the character to the left of the cursor.

If you backup again and look at the Rubout procedure you can see that it checks to make sure there really is a character to the left of the cursor. If so, it calls Remove to remove the character from the buffer. If not, it just makes the terminal Beep at the user so all his coworkers will know he did something stupid.

Suppose someone gave you the Get_Response subunit and asked you to draw a picture of it, using your favorite form of structure chart. The top level diagram would show what happens in the main sequence of statements. One of the items drawn under Get_Response would be Process_Character. If you drew a diagram of Process_Character, it would have Rubout hanging from it. If you drew a diagram of Rubout it would have Remove and Beep under it.

In general, the closer you get to the top of the listing, the closer you get to the bottom of the structure chart. This even extends to the context clauses because they name utility packages and subprograms that appear at the very bottom of the structure chart (if they appear at all).

3.6. FORM_TERMINAL package

Most of the programs I write use SCROLL_TERMINAL as the user interface, but there are some instances where I need some special features the SCROLL_TERMINAL doesn't have. Those applications need a FORM_TERMINAL.

The FORM_TERMINAL package is more sophisticated than the SCROLL_TERMINAL. In some instances it is a much easier interface for the user to use, but it puts a little more burden on the programmer. This is an unavoidable consequence of the law of conservation of energy. It takes a certain amount of work to get a job done. Some of the work has to be done by the user, and some has to be done by the programmer. The more work the programmer does, the less the user has to do, and vice versa. A programmer has to work hard to come up with an easy user interface.

The complete FORM_TERMINAL consists of nineteen listings. That's too much to tackle at once, so lets begin by looking at the first six listings (25 through 30). They provide most of the functionality of the FORM_TERMINAL. The last thirteen listings (31 through 43) are only used to create and modify forms, and will be discussed later.

3.6.1. When to use the FORM_TERMINAL

The FORM_TERMINAL is useful for those applications where a sequential data entry interface (such as the SCROLL_TERMINAL) is not appropriate. For example, suppose you used the SCROLL_TERMINAL for an income tax program. The SCROLL_TERMINAL would force the tax payer to answer far too many questions in a prescribed order. Suppose the tax payer got to Line 31 and realized he had made a mistake on Line 14. It is too late to go back and fix it. He would have to start all over and answer all the questions again. The taxpayer needs a way to go back and change an answer. The FORM_TERMINAL allows him to do that because he can fill in the blanks in any order. The data is not passed to the application program until the user is satisfied with the way the form looks.

I saw another excellent example of the good use for the FORM_TERMINAL when I went to see my insurance agent a few days ago. I wanted to make some changes on my car insurance. He sat down at his computer and typed in my name, and a screen full of information appeared. It showed my name, address, birthday, vehicle type, coverage limits, and who knows what else. The agent was able to move the cursor to the proper field, make the change to the coverage I wanted, and pressed a button. The computer figured my new premium. If he had used the SCROLL_TERMINAL to do that he would have had to have reentered my name, address, birthday, and so on. It would have been a real pain.

3.6.2. Consistency

Despite its advantages in situations like these, the FORM_TERMINAL isn't as good as the SCROLL_TERMINAL for user dialogs. Because of this, programs that use FORM_TERMINAL often use SCROLL_TERMINAL, too. (The FORM_TERMINAL examples you are about to see use the SCROLL_TERMINAL for help messages.) It would be terribly frustrating if the FORM_TERMINAL wasn't consistent with the conventions already established for the SCROLL_TERMINAL. The editing keys must work the same for both interfaces. If they don't the user will wonder why the INSERT key works some times and not other times.

All of the keys that work for SCROLL_TERMINAL work exactly the same way for the FORM_TERMINAL. The FORM_TERMINAL also uses four more keys. The UP and DOWN arrows mean "previous form" and "next form." The TAB and BACK_TAB keys mean "next field" and "previous field."

It was tempting to use some of the IBM PC keys, like "Pg Up", "Pg Dn", "Home", and "End" for the FORM_TERMINAL. I choose not to because some terminals, like the Televideo 910, don't have these keys. I wanted to keep the mental re- mapping of keys down to a minimum. I didn't want to have to remember that function key F5 is Pg Dn on a Televideo 910.

No matter what hardware is used, the FORM_TERMINAL always works the same, because it is built on the VIRTUAL_TERMINAL. No modifications are required to port it to an environment that already has the VIRTUAL_TERMINAL package running. That's good, because the FORM_TERMINAL is extensive, and makes heavy use of cursor control keys. It would be difficult to rehost if it was built on host operating system calls.

3.6.3. Abstract Objects

The FORM_TERMINAL is an example of a package representing a single object. It is an electronic representation of a special piece of paper called a form. Let's describe the properties of a form and see how they are represented in Ada.

I'm sure you've had experience with all kinds of forms. For example, income tax forms, employment applications, an application for a driver's license, and so on. What do all these forms have in common?

First, they all have a limited size. There are only so many characters you can fit on a single page. If all the information won't fit on a single page, you need a multiple- page form. In the FORM_TERMINAL package specification (Listing 25), I've defined the size of a single page with the subtypes Line_numbers and Column_numbers. This size is related to the size of the terminal screen because you need to be able to see the whole page all at once. I've made the form one line shorter than the number of lines on the screen so I can use the bottom line for status and instructions that aren't part of the form. It may be that you can't fit all the information you need on a single form, but that happens with paper forms, too. The solution is the same for the electronic form. Use more pages if necessary.

The second thing you notice about a form is that it is divided into parts. Some of these parts have information preprinted on them. Other parts are blank so the user can enter data. The FORM_TERMINAL calls these parts "fields." Protected fields are those printed parts the user can't change. Unprotected fields are the blanks he can fill in. Unlike paper forms, this electronic form can have a default response in an unprotected field.

Paper forms sometimes have numbered fields. That's because you sometimes need to refer to a particular field. In version 1, I numbered the fields on the form, but I found it hard to remember what numbers referred to each field. In version 2, I chose to give every field a 20-character name. The FORM_TERMINAL uses a subtype Field_names to represent the names of the fields.

Clearly there will be at least two things we will want to do with these fields. We will want to print instructions and default responses in some of them, and we will want to read what the user has written in the blanks. The get and put procedures allow us to do this. We can put a text string to any field we name, or we can get a string from any named field.

There are other, not so obvious, things we need to do with a form. One is to simply display it. More often we want to display the form and give the user a chance to update the information on it. The Display and Update procedures let us do that.

The two parameters CURSOR_AT and NEXT in the Update procedure need a little explanation. CURSOR_AT lets us place the cursor in any unprotected field. This is the field we expect the user to want to change first. Generally it will be the first unprotected field in the upper left corner of the form. Each time the user presses the RETURN or TAB key, the cursor will move to the next unprotected field. If the user presses the BACK_TAB key, the cursor moves to the previous unprotected field. The cursor will never appear in a protected field because protected fields contain things the user is not allowed to change.

When the user moves the cursor off the bottom of the screen (using the RETURN or TAB key in the last field, or using the DOWN arrow anywhere on the form), the Update procedure knows the user is finished with this form and wants to go on to the next page (if any). When this happens, the NEXT parameter is TRUE. It may be, however, that the user wants to go back to a previous page. He can do this by pressing the UP arrow, or using the BACK_TAB key until he moves the cursor off the top of the page. In this case Update returns with the NEXT parameter set to FALSE. This should be interpreted as a request to go back to the previous form and update it again.

This idea of multiple pages brings up an interesting design decision. How do you keep track of multiple page forms? In version 1, I had a limited private type called Forms and declared arrays of Forms. All the Forms were in memory at once, and I could move forward and backward by simply incrementing or decrementing the index. The problem was that I was using a less efficient internal representation of the form than I use now, and I could only get three forms in memory at once. Furthermore, version 1 of the Alsys compiler didn't realize it didn't have enough memory for a fourth form, and wrote it over my code.

The representation of forms used in version 2 (although the most compact form possible) takes much less space than version 1 needed, so I suspect I could hold as many forms in memory as necessary under normal circumstances. I was afraid, though, that someday I would have an application that required more forms than there was room for. This could happen because I needed lots of forms, or because the code segment of the program used up almost all of the memory space. I decided it was safer to keep just one form in memory at a time, and keep the others on disk. (This has a side benefit. If there is a power failure I lose only the information on the form currently being updated.)

The decision to store forms on disk required Read and Write procedures to be included in the package. This means the application program needs to know the names of the files containing the forms. I have a generic FILE_SYSTEM package that hides path names and makes it easy to port programs to different operating systems, but it is too complicated to include in an intermediate level book. It is not necessary to use a special file system, however. You can use a simple file name. (You will see how this is done in a later figure.)

There are several things that could go wrong when using the FORM_TERMINAL. You might try to read a file that doesn't exist, write a form to a full disk, get a form from a file that doesn't exist, or read a form from a file that doesn't really contain a form. The exceptions READ_ERROR, WRITE_ERROR, ASSIGNMENT_ERROR, and LAYOUT_ERROR are raised in these situations. With the exception of

```
Figure 21. Usual_Dilemma.
----------------------------------------------------
-- A beginning Ada programmer sometimes finds
-- himself in this dilemma.

with SCROLL_TERMINAL; use SCROLL_TERMINAL;
procedure Usual_Dilemma is

  NAME           : string(1..30);
  ADDRESS        : string(1..40);
  CITY_STATE_ZIP : string(1..40);

begin

  get("What is your Name? ", NAME);
  get("What is your Address? ",ADDRESS);
  get("Where do you Live? ",CITY_STATE_ZIP);

  SCROLL_TERMINAL.new_line(10);
  SCROLL_TERMINAL.put_line
    ("NAME = " & NAME);
  SCROLL_TERMINAL.put_line
    ("ADDRESS = " & ADDRESS);
  SCROLL_TERMINAL.put_line
    ("CITY = " & CITY_STATE_ZIP);

exception
  when NEEDS_HELP =>
    null; -- nothing you can do now!
end Usual_Dilemma;
```

WRITE_ERROR (disk full), these exceptions probably won't happen after you have debugged your application program.

There are two other exceptions that could happen under normal circumstances. These are PANIC and NEEDS_HELP. Both of these exceptions are raised at the user's whim, and no amount of debugging can prevent them. The PANIC exception is raised whenever the user says to himself, "Oops! I didn't want to do this. Let's quit." The NEEDS_HELP exception is raised whenever the user presses the question-mark key because he doesn't know how to answer the question. The NEEDS_HELP situation is a classic example of a puzzling problem to many new Ada programmers, so let's digress for a moment and talk about it.

3.6.4. Exception Handling

The common complaints about Ada's exceptions are (1) the exception does not include a status code telling the type of error, and (2) it is not possible to return to the place the exception was raised. These aren't really problems. They are simply features Ada doesn't need. People who criticize Ada for these "deficiencies" could be compared to sailors criticizing an automobile for not having a sail and a bilge pump.

People who don't know how to use Ada exceptions usually find themselves in the dilemma shown in Figure 21. This example lets the user enter a name and address and echoes it back. There is a possibility that the user will raise the NEEDS_HELP exception by pressing the question mark key. When this happens, the dilemma is that the NEEDS_HELP exception doesn't tell us which of the three questions confused the user, and we couldn't get back to that question even if we knew where we wanted to go.

The usual solution is to structure the program as shown in Figure 22. By encapsulating each query in a separate procedure it is possible to separate the exception handlers. Each exception handler gives an appropriate help message and then recursively calls the appropriate input routine. This is a good solution

Figure 22. Usual_Solution.

```
-------------------------------------------------------------
-- The usual solution is to avoid the problem
-- by handling the exception where it occurs.

with SCROLL_TERMINAL; use SCROLL_TERMINAL;
procedure Usual_Solution is

  NAME          : string(1..30);
  ADDRESS       : string(1..40);
  CITY_STATE_ZIP : string(1..40);

  procedure Get_Name(NAME : out string) is
  begin
    get("What is your Name? ", NAME);
  exception
    when NEEDS_HELP =>
      put_line("Don't you even know your own name?");
      Get_Name(NAME);
  end Get_Name;

  procedure Get_Address(ADDRESS : out string) is
  begin
    get("What is your Address? ", ADDRESS);
  exception
    when NEEDS_HELP =>
      put_line("What is your street or P.O Box?");
      Get_Address(ADDRESS);
  end Get_Address;

  procedure Get_City(CITY : out string) is
  begin
    get("Where do you Live? ", CITY);
  exception
    when NEEDS_HELP =>
      put_line("Please enter City, State, and Zip code.");
      Get_City(CITY);
  end Get_City;

begin

  Get_Name(NAME);
  Get_Address(ADDRESS);
  Get_City(CITY_STATE_ZIP);

  SCROLL_TERMINAL.new_line(10);
  SCROLL_TERMINAL.put_line
    ("NAME = " & NAME);
  SCROLL_TERMINAL.put_line
    ("ADDRESS = " & ADDRESS);
  SCROLL_TERMINAL.put_line
    ("CITY = " & CITY_STATE_ZIP);

end Usual_Solution;
```

because it keeps the exception handler close to the point where the exception will be raised, and it keeps the error routines from cluttering the main program. I recommend using this solution whenever possible.

Unfortunately there are cases where this solution won't work. We are faced with such a situation when we use the FORM_TERMINAL instead of the SCROLL_TERMINAL to get the name and address. Consider the Form_Dilemma shown in Figure 23. ADDRESS.DAT is a file containing the data necessary for drawing a simple name and address form on the screen. We will look at the contents of this file in a few pages. Right now all you need to know is that it defines a form that prompts for name, address, and city, and tells where these fields should appear on the screen. Form_Dilemma fetches the blank form from ADDRESS.DAT, lets the user update it, extracts the name and address from the form, and echoes it to the

```
Figure 23. Form_Dilemma.
------------------------------------------------------------
-- In this case it wouldn't do any good to try to give
-- Update a local exception handler because you don't know
-- what field the user was updating when he requested help.

with SCROLL_TERMINAL;
with FORM_TERMINAL; use FORM_TERMINAL;
procedure Form_Dilemma is

  NAME            : string(1..30);
  ADDRESS         : string(1..40);
  CITY_STATE_ZIP  : string(1..40);
  DOESNT_MATTER   : boolean;

begin

  Read("ADDRESS.DAT");
  Update(CURSOR_AT => "Name field          ",
              NEXT => DOESNT_MATTER);

  get("Name field          ", NAME);
  get("Address field       ", ADDRESS);
  get("City field          ", CITY_STATE_ZIP);

  SCROLL_TERMINAL.New_Line(10);
  SCROLL_TERMINAL.put_line("NAME = " & NAME);
  SCROLL_TERMINAL.put_line("ADDRESS = " & ADDRESS);
  SCROLL_TERMINAL.put_line("CITY = " & CITY_STATE_ZIP);

exception
  when NEEDS_HELP =>
    null; -- nothing you can do now!
end Form_Dilemma;
```

screen. It works fine, unless the user NEEDS_HELP. Then we are back to another variation of the usual dilemma. We are in the exception handler, don't know why, and don't know how to get back. What are we to do?

A horrible solution is shown in Figure 24. We will see a much better way to solve the problem in a moment, but we must suffer through this bad example just to see what's wrong with it. I call this the FORTRAN_Mentality_Solution because FORTRAN teaches people to program this way. Some Ada critics claim you HAVE to solve the problem this way. If that was true, they would be justified in their criticism. But let's not damn Ada for their ignorance.

The FORTRAN_Mentality_Solution is to rewrite Update somehow to eliminate the NEEDS_HELP exception and replace it with a STATUS variable. Every time the procedure is called you must check the STATUS variable to see if the procedure completed correctly. There are two problems with this. First, it forces you to depend upon every application programmer who will ever use the Update procedure to remember (or care enough) to check the STATUS variable. Second, it adds overhead every time you use it, to make sure the procedure completed correctly.

The FORTRAN_Mentality_Solution requires several GOTO statements. Ada doesn't normally need GOTOs. The GOTO is included in the LRM for no other reason than to allow you to convert poorly structured FORTRAN into poorly structured Ada. That's what I've done here. Please don't consider this an endorsement of GOTOs. Remember, this is the wrong way to solve the problem.

The right way to solve the problem is shown in Figure 25. It is similar to the usual solution because it uses a block structure to encapsulate a routine that is likely to raise an exception, and provides a local exception handler for the routine. This eliminates the need to go back to the point of the exception because we

```
Figure 24. FORTRAN_Mentality_Solution.
----------------------------------------------------
-- This is the wrong way to solve the problem.

with SCROLL_TERMINAL;
with FORM_TERMINAL; use FORM_TERMINAL;
procedure FORTRAN_Mentality_Solution is

  NAME            : string(1..30);
  ADDRESS         : string(1..40);
  CITY_STATE_ZIP  : string(1..40);
  DOESNT_MATTER   : boolean;
  HELP_REQUEST    : integer;
  CURSOR_POSITION : Field_names;

begin
  Read("ADDRESS.DAT");
  CURSOR_POSITION := "Name field          ";

  << ASK_NAME_AND_ADDRESS >>
  Update(CURSOR_AT => CURSOR_POSITION,
              NEXT => DOESNT_MATTER,
            STATUS => HELP_REQUEST);
  if HELP_REQUEST = 0 then
    null; -- no help needed
  elsif HELP_REQUEST = 1 then
    SCROLL_TERMINAL.new_line(10);
    SCROLL_TERMINAL.put_line
      ("Don't you even know your own name?");
    SCROLL_TERMINAL.Wait_For_User;
    CURSOR_POSITION := "Name field          ";
    goto ASK_NAME_AND_ADDRESS;
  elsif HELP_REQUEST = 2 then
    SCROLL_TERMINAL.new_line(10);
    SCROLL_TERMINAL.put_line
      ("What is you street or P.O. Box?");
    SCROLL_TERMINAL.Wait_For_User;
    CURSOR_POSITION := "Address field       ";
    goto ASK_NAME_AND_ADDRESS;
  elsif HELP_REQUEST = 3 then
    SCROLL_TERMINAL.new_line(10);
    SCROLL_TERMINAL.put_line
      ("Where do you live?");
    SCROLL_TERMINAL.Wait_For_User;
    CURSOR_POSITION := "City field          ";
    goto ASK_NAME_AND_ADDRESS;
  end if;

  get("Name field          ", NAME);
  get("Address field       ", ADDRESS);
  get("City field          ", CITY_STATE_ZIP);

  SCROLL_TERMINAL.new_line(10);
  SCROLL_TERMINAL.put_line("NAME = " & NAME);
  SCROLL_TERMINAL.put_line("ADDRESS = " & ADDRESS);
  SCROLL_TERMINAL.put_line("CITY = " & CITY_STATE_ZIP);
end FORTRAN_Mentality_Solution;
```

haven't really left the point of the exception. In this case, that's only half the solution. We still have to figure out why NEEDS_HELP was raised.

Ada doesn't provide any way for me to pass the WORKING_FIELD number back along with the exception. Even if she did, it wouldn't do me much good because the application program thinks in terms of the names of the fields, doesn't know the fields are in an array indexed by an integer, and doesn't know WORKING_FIELD is the index. (If I let the application programs know this, I don't dare ever change the representation of a FORM for fear it will mess up a critical application program someone else wrote.)

Figure 25. Form_Solution.

```
---------------------------------------------------------
-- The right way to solve the problem, using a local
-- exception handler and a function that returns
-- additional error information.

with SCROLL_TERMINAL;
with FORM_TERMINAL; use FORM_TERMINAL;
procedure Form_Solution is

  NAME           : string(1..30);
  ADDRESS        : string(1..40);
  CITY_STATE_ZIP : string(1..40);

  procedure get(FIRST_FIELD : Field_names;
        NAME, ADDRESS, CITY : out string) is
              DOESNT_MATTER : boolean;
  begin
    Update(CURSOR_AT => FIRST_FIELD,
                NEXT => DOESNT_MATTER);
    get("Name field          ", NAME);
    get("Address field       ", ADDRESS);
    get("City field          ", CITY_STATE_ZIP);
  exception
    when NEEDS_HELP =>
      SCROLL_TERMINAL.new_line(10);
      if Confusing_Field =  "NAME FIELD          " then
        SCROLL_TERMINAL.put_line
          ("Don't you even know your own name?");
      elsif Confusing_Field =  "ADDRESS FIELD       " then
        SCROLL_TERMINAL.put_line
          ("What is your street or P.O Box?");
      elsif Confusing_Field =  "CITY FIELD          " then
        SCROLL_TERMINAL.put_line
          ("Please enter City, State, and Zip code.");
      end if;
      SCROLL_TERMINAL.Wait_for_User;
      get(Confusing_Field, NAME, ADDRESS, CITY_STATE_ZIP);
    end get;

  begin

    Read("ADDRESS.DAT");
    get("Name field          ",NAME, ADDRESS,
      CITY_STATE_ZIP);

    SCROLL_TERMINAL.New_Line(10);
    SCROLL_TERMINAL.put_line("NAME = " & NAME);
    SCROLL_TERMINAL.put_line("ADDRESS = " & ADDRESS);
    SCROLL_TERMINAL.put_line("CITY = " & CITY_STATE_ZIP);
  end Form_Solution;
```

Application programs using the FORM_TERMINAL will know about Field_names because they use them to get and put data, and position the cursor. The application programmer needs to know the name of the field being processed when the NEEDS_HELP exception raised. The real key to the solution is having the foresight to include the Confusing_Field function in the FORM_TERMINAL package. Whenever the NEEDS_HELP exception is raised, the application program can call the Confusing_Field function to find out the name of the field that was being processed when the exception was raised. This is similar to checking a status variable, but the difference is that you only do it on those rare occasions when the exception occurs.

To do this I had to move the WORKING_FIELD number out of Update (where nobody but Update can use it) and put it in the FORM_TERMINAL body. Here other FORM_TERMINAL subprograms have access to it. The WORKING_FIELD variable is shared by Update and Confusing_Field. Update reads and writes it. Confusing_Field reads it and converts it to the corresponding NAME, which is what the application program needs to know.

The Confusing_Field function not only gives the application program all the information necessary to handle the exception, it also leaves me free to change the internal representation of the FORM. Suppose there was a compelling reason for me to change to a linked list (which uses an access type FIELD_POINTER instead of the integer WORKING_FIELD) to represent a FORM. I could do this with full assurance that I wouldn't have to rewrite any application programs that use FORM_TERMINAL, providing I put FIELD_POINTER in the package body, and I modify the Confusing_Field function to convert FIELD_POINTER to a field name.

3.6.5. Keep Shared Variables Hidden

You might be tempted to do the same thing by declaring the shared variable in the package specification. Then you wouldn't need a subprogram to get at it. That's asking for trouble because who knows what stupidity lurks in the mind of the application programmers who will come after you. Somebody might assign it to 0 before calling your routines, check it on completion to see if it is 0, and call an error routine if it isn't. Maybe the initial value of 0 will mess up your routine. Maybe your routine normally changes it, even if there isn't an error. Protect yourself (and your good name) from them. Keep the variable hidden in the body and let others use it only in a controlled way using routines you have written yourself.

There is another reason for keeping the shared variable out of the package specification. If the shared variable is in the package specification, you have lost the ability to change internal representations. Suppose WORKING_FIELD was in the package specification, and you changed to a linked- list scheme that uses FIELD_POINTER. Then every application program that used WORKING_FIELD wouldn't work any more, and would have to be rewritten.

So, the general lesson is this: Whenever you have information you need to pass back to an application program when something goes wrong, store that information in a variable declared in the package body, instead of a subprogram body. Then write another subprogram in that same package that can return the information to the application program in the most useful form.

3.6.6. Reuse by Copy

You may have noticed that the Get_Form subunit (Listing 27) is strikingly similar to the Get_Response subunit (Listing 24) in the SCROLL_TERMINAL. These two routines are too different to be derived from a common generic unit, but too similar to start completely from scratch. I simply made a copy of Get_Response and used a text editor to change it a little to create Get_Form.

There are major differences between the two units. You can find them easily using a file comparison utility program. Most of the differences aren't worth much discussion. The FORM_TERMINAL doesn't need to check the ECHO flag because it isn't designed to be used in applications where it shouldn't echo the response. It doesn't have to worry about how many characters the user has entered because the SIZE of the blank of the form is constant. But even though there were significant differences, I still saved time and effort by editing a copy of an existing unit instead of starting from scratch.

The difference that is worth talking about in detail has to do the parameter list. Get_Response passes DEFAULT, TEXT, and LENGTH as parameters. When the user presses the RETURN key he is done, and that's all there is to it. The FORM_TERMINAL is more complicated because the user can end his response by saying he wants to go to the NEXT_FIELD, PREVIOUS_FIELD, NEXT_FORM, or PREVIOUS_FORM. That explains why Get_Form has to return one of those Actions. The parameter list of Get_Form clearly shows this, but it isn't immediately obvious how Get_Form knows what the prompts and defaults are, or how it returns the user's responses.

3.6.7. Global Variables

The Get_Form reads data from, and writes data to a (dare I say it?) global variable called FORM. I know how some of you feel about global variables. I feel the same way. It is usually a bad idea to use a global variable because it obscures the coupling between modules. If several variables write to the same global variable, and it is found to contain the wrong value, it is sometimes hard to determine which is the guilty module. Suppose two modules communicate through a global variable, and you decide to change the meaning of the value in both of those modules, you will get unusual, difficult to locate errors if you have forgotten that a third module also uses that variable.

You don't see me using global variables very often. In this case, however, I feel justified in using them. True, I could pass an object of type Forms, but that seemed awkward. It wouldn't explicitly show the prompts, defaults, and response, so it didn't really provide any more information than using the global variable. Since there is only one object of type Forms, I can't possibly get it confused with any other object of the same type. The application program can't see FORM, so it can't corrupt it.

The FORM is indexed by WORKING_FIELD, which I definitely don't want to pass as a parameter because there is a possibility that an exception might be raised. (If WORKING_FIELD is passed as a parameter I can't guarantee it will be the correct value if NEEDS_HELP is raised.) It seems strange to me to pass FORM as a parameter but index it with a global variable. The exception argument also holds for the FORM itself. If the user NEEDS_HELP, I can be sure the global variable FORM contains the user's partially edited form. If FORM is passed back as a parameter, and the user NEEDS_HELP, there's no telling what is in FORM. Maybe it is the virgin form before the user started editing it. Maybe it is the partially edited form. Maybe it is garbage pointed to by whatever number happened to be in the stack frame when the exception was raised. This is just one of those rare cases where a global variable makes more sense than a passed parameter.

3.6.8. A Package Can be an Abstract Object

Perhaps the most important concept in Listing 26 is the data type Forms. By declaring Forms in the body rather than the specification, I have made it even more private than limited private. Application programs can declare objects of type limited private, but they can't even declare objects of type Forms because it isn't visible to them. If they can't declare objects of type Forms, what good is it? Plenty, but you may have to change the way you think about objects.

You are probably used to packages that contain objects. This time the package itself is an object. The package represents something that has a value. When you write FORM_TERMINAL.put(SOME_FIELD,"Some Text"); you are actually assigning a value to a component of that object. You can read it back using the statement FORM_TERMINAL.get(SOME_FIELD,STRING_VARIABLE);. That's why you don't need to declare objects of type Forms. The package itself is the form.

3.6.9. Discriminated Records

Type Forms is a discriminated record with three components. It consists of a number of fields, an array of that number of fields, and an image of the screen. We will soon see that discriminated records are a little difficult to work with, but they solve an important problem. They avoid the need to guess how many fields there will be on the form.

Looking at the FIELD component, we see that it is an array of Field_specs, where each specification tells the name of the field, what line it is on, where it begins and ends, and if it is protected or not. The text showing on the form is not part of the field specification. Instead, it is stored in a two dimensional array of characters called a SCREEN. Since many of the characters are blank, there is a potential for saving some

Figure 26. Objects need a constraint.

```
---------------------------------------------------------
-- This procedure illustrates the consequences of LRM
-- Section 3.7.2 paragraph 8.

procedure Objects_Need_A_Constraint is

  type No_default (LENGTH : natural) is
    record
      TEXT : string(1..LENGTH);
    end record;

  type Default (LENGTH : natural := 0) is
    record
      TEXT : string(1..LENGTH);
    end record;

  FIRST_NAME : No_default(8);
  MY_NAME    : No_default;    -- illegal
  WHOLE_NAME : Default(14);
  LAST_NAME  : Default;

  -- MY_NAME is illegal because no constraint was
  -- supplied (as it was for FIRST_NAME).

  -- LAST_NAME doesn't need a constraint because there
  -- is a default value for it.

  -- WHOLE_NAME shows that you can give a constraint
  -- that is different from the default value if you like.

begin
  null;
end Objects_Need_A_Constraint;
```

space by storing text in the Field_specs and doing away with the SCREEN. (This requires nested discriminated records, and I didn't want to get that complicated.)

Version 1 of the FORM_TERMINAL didn't use a discriminated record. It used an ordinary record, and one component of the record was a fixed length array of Field_specs. A constant MAX_FIELDS set the size of this array. At various times this constant was 30, 60, or 100. When I tried using 30 I ran into trouble because I often wanted more than 30 fields on the form. When I tried 100 fields, it used up so much memory I only had room for one form in memory. I finally settled on 60 as a reasonable compromise, but I always worried it would be too large or too small.

The discriminated record lets me declare an array of fields that is exactly the right size. The problem is that I have to know how many fields will be on the form before I declare it, and I have to change the entire form in one shot if I want to redimension it. That's not a trivial problem, but it is far from impossible. If you have worked with discriminated records before you have probably run into this difficulty. Maybe you gave up. If you did, you'll be glad to see this solution.

For those of you who have not run into this problem, let me quote three pertinent parts of the LRM.

> For a variable declared by an object declaration, the subtype indication of the corresponding object declaration must impose a discriminant constraint unless default expressions exist for the discriminant. (Section 3.7.2 paragraph 8)
>
> If the type of an object is a type with discriminants and the subtype of the object is constrained, the implicit initial (and only) value of each discriminant is defined by the subtype of the object. (Section 3.2.1 paragraph 12)

> Direct assignment to a discriminant of an object is not allowed; ... The only way to change the value of a discriminant of a variable is to assign a (complete) value to the variable itself. (Section 3.7.1 paragraph 9)

Here are three examples to show what those three parts of the LRM mean in practice. I've written these examples as procedures so you can compile them to see what error messages your compiler generates.

Figure 26 shows that an object must be constrained when you declare it. This constraint can be explicitly shown, as in FIRST_NAME or WHOLE_NAME, or it can be the default, as shown in LAST_NAME. MY_NAME is illegal because it has no explicit constraint and it has no default constraint.

Figure 27 shows that constraints can be changed only when the object was declared with a default constraint. Since LAST_NAME was declared without an explicit constraint, that constraint can be changed. Notice that it is the object declaration, not the type definition, that counts. WHOLE_NAME is of type Default, so it has a default constraint, but we didn't use the default when we declared it. When we declared WHOLE_NAME : Default(14); we told Ada we want WHOLE_NAME to always have a 14 character TEXT string, and she takes us at our word. If we change our mind later, its just tough luck.

Figure 27. Constrained objects can't change constraints.

```
---------------------------------------------------------
-- This procedure illustrates thee consequences of LRM
-- Section 3.2.1 paragraph 12.

procedure Constrained_Objects_Cant_Change_Constraints is

  type No_default (LENGTH : natural) is
    record
      TEXT : string(1..LENGTH);
    end record;

  type Default (LENGTH : natural := 0) is
    record
      TEXT : string(1..LENGTH);
    end record;

  FIRST_NAME : No_default(8);
  WHOLE_NAME : Default(14);
  LAST_NAME  : Default;

begin

  -- I can let LAST_NAME hold a 5 character name.
  LAST_NAME := (LENGTH => 5, TEXT => "Jones");

  -- Then I can stretch it to hold a 10 character name.
  LAST_NAME := (10, "Washington");

  -- FIRST_NAME can hold an 8 character name.
  FIRST_NAME := (8, "Do-While");

  -- But it can't shrink to hold a shorter one
  -- because it was constrained to be 8 characters
  -- when it was declared.
  FIRST_NAME := (4, "Dave"); -- ILLEGAL

  -- You might think WHOLE_NAME can be changed because
  -- it has a default length, but the fact is it was
  -- constrained to be 14 characters when it was
  -- declared, so it can't be changed either.
  WHOLE_NAME := (17, "George Washington"); -- ILLEGAL

end Constrained_Objects_Cant_Change_Constraints;
```

The moral of the story so far is, "If you want to be able to change a discriminant of an object, the type definition must include a default value and the object declaration must use that default."

Finally Figure 28 shows us that even under the special circumstance when we can change the discriminant, we may only change it in a particular way. We can't change the discriminant alone, we must change every component of the object at once. The only way to do that is with an assignment statement. We can assign the value of another object of the same type to it, or we can assign an aggregate (but not a slice) to it.

The rest of the lesson is, "You either set the discriminant to the the correct (constant) size when you declare the object, or declare the object without a constraint and change the whole object at once." The two ways to change an object all at once are to assign all the components using an aggregate, or assign it to another object of the same type (even though it has a different constraint).

Of course there was a reason for this long explanation. The FORM_TERMINAL needs to read a discriminated record from a disk file. In general, this means it needs to change the size of a discriminated record currently in memory, and read new data into it. The preceding discussion was designed to impress upon you how tricky this is.

Figure 28. You can't change the constraint alone.

```
------------------------------------------------------------
-- This procedure illustrates the consequences of LRM
-- Section 3.7.1 paragraph 9.

procedure Cant_Change_Constraint_Alone is

  type Default (LENGTH : natural := 0) is
    record
      TEXT : string(1..LENGTH);
    end record;

  WHOLE_NAME : Default(14);
  LAST_NAME  : Default;

begin

  -- I can let LAST_NAME hold a 5 character name.
  LAST_NAME := (LENGTH => 5, TEXT => "Jones");

  -- Then I can change those five characters alone.
  LAST_NAME.TEXT := "Smith";

  -- But I can't change the discriminant alone.
  LAST_NAME.LENGTH := 10; -- ILLEGAL

  -- WHOLE_NAME was constrained to be 14 characters,
  -- so I can do this:
  WHOLE_NAME.TEXT := "Do-While Jones";

  -- And I can expand LAST_NAME by changing both the
  -- LENGTH and TEXT in one shot by an assignment
  -- statement.
  LAST_NAME := WHOLE_NAME;
  -- (LAST_NAME now contains (14,"Do-While Jones").)

  -- Or I can assign the whole record using an
  -- aggregate, like this:
  LAST_NAME := (5, "Jones");

end Cant_Change_Constraint_Alone;
```

3.6.10. Reading Discriminated Records from a File

Data is stored in a file similar to the one shown in Figure 29. This is the ADDRESS.DAT file we first saw in the Form_Dilemma program (Figure 23). The first line of the file contains the number 7 because there are seven fields on the form. Each of the seven field specifications begin with the header "-- data --". The six lines following each header tell (1) the name of the field, (2) the line the field appears on, (3) the first column and (4) last column of the field, (5) whether it is protected or unprotected, and (6) the text that appears in the field. You can't tell it from the figure, but the field names have been padded with spaces to make them exactly twenty characters long. The lines that appear to be blank actually contain the number of blank spaced to fill the first through last columns of the field.

```
Figure 29. ADDRESS.DAT
--------------------------------------------------------
 7
-- data --
FORM TITLE
 4
 3
 23
P
Name and Address Form
-- data --
NAME PROMPT
 7
 1
 20
P
Name               :
-- data --
NAME FIELD
 7
 22
 51
U

-- data --
ADDRESS PROMPT
 8
 1
 20
P
Address            :
-- data --
ADDRESS FIELD
 8
 22
 61
U

-- data --
CITY PROMPT
 9
 1
 20
P
City, State, & ZIP :
-- data --
CITY FIELD
 9
 22
 61
U

[end of file]
```

The Read subprogram (Listing 28) opens the input file and reads the first line. This line contains the number of fields in the FORM. Then it calls a function, Stored_Form, passing it the number of fields as a parameter. The function Stored_Form declares and object TEMP of type Forms, with the discriminant set to the proper size. It reads the information from the file a line at a time, and builds up TEMP a piece at a time (but it never changes the discriminant because it was the correct value to begin with). When it is all assembled, TEMP is returned as the result and assigned to FORM all at once. So, reading a discriminated record from a file isn't difficult, if you know how to do it.

3.6.11. Discriminants May Not Save Space

The motivation for using a discriminated record was to save space. If there are only three fields on the form, why bother to define Field_arrays big enough to hold 100 Field_specs? In theory we could save memory space by using a discriminated record, declaring Field_arrays to hold only as many Field_specs as necessary. In practice it may not work that way. The LRM permits an implementation to allocate enough space to accommodate the maximum possible discriminant. The discriminated record may actually take more space than an ordinary record with a few empty fields. For example, an early version of Listing 26 declared Forms this way:

```
type Forms(FIELDS : positive := 1) is
          record
               FIELD  : Field_arrays(1..FIELDS);
               SCREEN : Screens;
          end record;
```

This worked on the Meridian Ada compiler, but raised an exception on the DEC Ada compiler, presumably because DEC Ada tried to allocate space for

```
FIELD : Field_arrays(1..positive'LAST);
```

and didn't have enough memory space to do it. I solved the problem by introducing a subtype which limits the maximum number of arrays.

```
subtype Field_numbers is positive range 1..200;
type Forms(FIELDS : Field_numbers := 1) is ...
```

This appears to be exactly the same as defining Forms using an ordinary record.

```
type Forms is
     record
          FIELDS : positive;
          FIELD  : Field_arrays(1..200);
          SCREEN : Screens;
     end record;
```

Both take exactly the same amount of memory space (on some implementations). The difference only appears in Listing 28. There you find the following statements:

```
function Stored_Form(SIZE : positive) return Forms is
     TEMP : Forms(SIZE);
```

I think that calling Stored_Form(10) will allocate space for ten Fields_arrays instead of two hundred, and therefore saves a little space, but I can't be sure of that.

Compilers vendors have taken different approaches to implementing discriminated records. You can't depend on every validated Ada compiler to implement discriminated records exactly the same way. This means you can't count on discriminated records to use the minimum required space, and that could cause portability problems if you use discriminated records.

If I had it to do all over again, I might not have used a discriminated record because of their unpredictable nature. I was tempted to rewrite the FORM_TERMINAL using an ordinary record and a generic parameter MAX_FIELDS, or perhaps use access types to build an unbounded linked list of Field_specs. I decided not to because it would have left me without an example of discriminated records.

3.6.12. Use Read as a Pattern for Write

Unless this is your first week on the job, I'll bet you've witnessed this scenario several times. Data from an important test had to be recorded in real time and analyzed immediately after the test. The project engineers carefully devised a format for recording the data on tape. The test was performed and data was recorded using that format. The tapes were taken to the data processing people and project management anxiously await the results. Weeks later nobody had seen any reduced data. The data processing people still had not figured out how to read the tapes yet! The project engineers blamed the data processing section. The head of the data processing department defended his people and tried to put the blame on the project engineers. Why does this happen?

It is easy to see how this could have happened with the routine to Read procedure. The first thing it does is read the number of Field_arrays from the first line of the file and create a discriminated record the proper size to hold the data. Suppose I hadn't written the number of Field_arrays to the first line of the file. I could have thought that it isn't necessary to waste a line on writing that information because it is possible to find the number of fields by reading the file and counting the number of times the "-- data --" header line appears. If I had done that, I would have had to read the whole file to find out how many field specifications it contains, create a form the proper size, and then read the file again to put the data in the form. (Or I could save all the data in memory from the first pass through the file, and then transfer it to the form if there is enough memory space.) If I had written the TEXT in each field before writing the FIRST and LAST column numbers. It would have been much more difficult to read the form from the file. If I had made either of these stupid decisions when I defined the file format, the data wouldn't have been lost. I still could have read the file with a little extra effort.

Writing a file is easy. You can write anything in any order without any problems. Reading a file can be difficult because you often need to know certain pieces of information before you can process others. So, the first rule for establishing a file format is, "Write the routine that gets the data from the file before you write the routine that puts it there." After you have done this you can use your Read routine as a pattern for your Write routine.

Look at Listing 28, which contains both the Read and Write procedures. Let's compare Read to Write. The first thing Read does is Open a file, so the first thing Write does is Create a file. The next thing Read does is to read the number of fields on the form, so Write must write the number of fields on the form before writing anything else. Read uses that number to set the limit on a loop that reads a header, NAME, LINE, FIRST, LAST, PROTECTED, and TEXT, so Write should also set up a loop that writes those things in that order. Read ends by closing the file, so Write should end that way, too. The only thing Read does that doesn't correspond to something Write does is the error checking. (That's because Read can't be sure the file it is trying to read really contains a valid form.)

The Read routine in Listing 28 is more complicated than most file reading routines because it needs the Stored_Form function to create a discriminated record. In many cases a file reading routine is a simple loop or straight line program. In those cases you can create a Write routine from a Read routine simply by using a text editor to change all occurrences of get to put and then make some other minor changes (change Open to Create, for example). But whether you copy and edit the Read source file, or just use a printed copy of the Read source code as a guide for writing the Write source code, the principle is the same. Pick a format that is easy to read, write the Read routine first, and then use that as a pattern for the Write routine.

3.6.13. ASCII Data Files

You perhaps noticed that I used ASCII (rather than binary) format for the FORM files. Not only that, I wastefully put only one data item on each line. Binary files generally use less disk space and are faster than ASCII files. In this case the ASCII files are probably small enough to fit in a single disk sector with room to spare, so an ASCII file probably isn't any bigger than a binary file. The time it takes to convert those few words from binary to ASCII and back is negligible. In cases like these I always prefer ASCII to binary files because I can easily display, print, and edit them.

If I try to write a FORM to a file and then read it back, and it doesn't work, how do I know what went wrong? Did it write the file correctly and fail to read it? or did it write the file incorrectly? It is easy to print an ASCII file and see. Of course there are utility programs to dump and patch binary files. You can examine blocks of hexadecimal listings and find out what data was written to the file, but that's not as easy as looking at ASCII files with one data item per line. So even if I know I'm eventually going to be dealing with huge files, I usually start developing the file IO routines with ASCII representations of dwarfed files. After they are debugged I switch to binary and test the routines again with small files. Then I try them with the big files.

A computer can easily count lines to determine which numbers are associated with each variable, but I can't. I had some difficulty figuring out which numbers represented lines and columns, especially if I was interested in a field specification near the middle of the file, so I added header lines that said "-- data --" at the beginning of each field specification. They stick out like a sore thumb, and make it easy for me to visually see where each field specification begins.

The Read program could ignore the "-- data --" lines, since they convey no information. This is easily done using skip_line. I decided not to skip them, but to use them as parity checks instead. Every time I would normally have skipped the header line, I read it and make sure it really says "-- data --". If it doesn't, it means the file has been corrupted, the Read routine has gotten out of sync, or the file doesn't really contain a form. In any of those cases I don't want to continue trying to read the form, so it raises the READ_ERROR exception and quits.

3.6.14. Storing Boolean Values in a File

I've seen a message on an electronic bulletin board saying that a particular implementation of TEXT_IO has a bug in it that prevents it from properly storing boolean types when ENUMERATION_IO is instantiated for boolean types. I'm not sure if that's true or not. (Perhaps that person just wasn't using it correctly.) What I am sure of is that they were trying to use ENUMERATION_IO to store a boolean value, and I don't think that's a good idea.

The Read and Write routines store the boolean variable PROTECTED in an external file without instantiating ENUMERATION_IO. They simply uses the character P to indicate protected fields and U to indicate unprotected ones. What could be easier?

If you really have you heart set on writing TRUE and FALSE, you can use the attributes boolean'IMAGE and boolean'VALUE to convert between Boolean values and text strings, just as I used integer'IMAGE and integer'VALUE to do the same for numbers. I don't see any reason to use four or five characters where one will do the job, but you may have a good reason. Consider this, however. When you look at the ASCII representation of the file, TRUE and FALSE don't tell you much. They just tell you something is true or false. If you see a TRUE in a file, does that mean the field is unprotected? You have to think about it. If I were going to use several characters instead of just one, I would write PROTECTED or UNPROTECTED to the file, not TRUE or FALSE.

3.6.15. One Compilation Unit Per File

By now you must have noticed that I almost always put just one compilation unit in a file. I could combined all 19 FORM_TERMINAL listings in a single file. That would have made it easier for you to compile the FORM_TERMINAL. You could just submit that one file to the compiler and go visit your coworkers at the water cooler. Ten minutes later you could return to your terminal and see if it was done yet.

There are two good reasons not to combine several compilation units in one file. The primary one is that you have to recompile a whole file at a time. If the file contains ten long compilation units, and you change one, then you waste time recompiling nine units that haven't changed.

The secondary reason is that separate files allow you to make it easier to find particular compilation units. If there is something wrong with the Display routine, it is easier to look in Listing 29 than to search a huge listing looking for it. (This reason for separating compilation units isn't as compelling as it once was because modern software engineering environments make it possible for anyone associated with the project to search any file file for anything electronically, but I still think it is a good idea to try to keep files small.)

Everyone who has completed an introductory Ada course should have had it pounded in his head why package specifications should be separated from package bodies. I shouldn't need to tell you that using separate files for the package body and specification allow you to make changes to the body without making units that depend on the specification obsolete. I won't insult your intelligence by reminding you of that.

You don't need to combine compilation units in a single file to compile them all at once. Every operating system has something equivalent to a shell script (perhaps a ".BAT" file, or ".COM" file) that lets you execute a sequence of commands at once. Whenever I have a software component like the FORM_TERMINAL that is spread out over several files, I just write a script that compiles them all in the correct order.

There are times, however, when you have to break the "One unit per file" rule. Some Ada compilers require all parts of a generic package or subprogram be in a single file. On those compilers you don't have any choice but to put multiple compilation units in the file in that case. Since I know that is a potential portability problem, I always put all the components for a generic unit in one file whether the compiler I am using requires me to or not.

3.6.16. Encapsulating Details in One File

Listing 28 is another example of when to break the rule. It contains two separate subunits, Read and Write, even though they aren't generic.

Normally, I try to encapsulate design details in a single compilation unit. The format of the file containing a FORM is a design detail I would like to confine to one location. If possible, I would like to make only one compilation unit dependent upon the external file format, so any changes to that format will require me to recompile only one unit.

In this case, Read and Write both need to know the external file format. Since the file format affects both Read and Write, any changes made to format affect both subunits. There isn't any practical way I can see to encapsulate the format in just one unit. If you change one subunit without changing the other, it will cause problems.

Ada's compilation order rules sometimes help out, but not this time. If you change the FORM_TERMINAL body, Ada will realize that Read and Write are obsolete and need to be recompiled; but since Read and Write are both subunits of the body, you can change and recompile either without making the other one obsolete. Ada won't automatically tell you that you have to change and recompile the other subunit.

Since I couldn't encapsulate the external file format in a single subunit, I did the next best thing. I encapsulated it in a single file. If I modify one of units, I'm bound to notice the other one and remember that it has to be changed, too. This isn't foolproof. It is possible to open the file, change the format in one subunit without changing the other, and recompile the file, but it's had for me to imagine someone who could do that accidentally.

Putting both subunits in the same file not only reminds me to make the same changes in both, it also makes it easier to use the text editor to cut and paste patches to both subprograms at once.

Encapsulating the external file format in a file with two subunits gives us the flexibility to change the external format without affecting any other part of the program. If we want to use binary external form instead of ASCII, we can change the Read and Write routines, and all our changes are confined to one source file. Whenever we compile that file, we automatically compile a matching pair of routines. We never have to worry about accidentally compiling the old ASCII format Read and the new binary format Write.

3.6.17. Formatted I/O

I've hated formatted output ever since I first encountered a FORMAT(F6.2) statement 22 years ago. You would think that after all these years it would have gotten easier, but it hasn't. It is still easy to make a mistake when counting spaces, so column headings don't line up correctly! I never seem to get it write the first time.

Laying out a two dimensional form is even more hassle than laying out one dimensional column headers. The FORM_TERMINAL requires you to count rows, columns, and string lengths. Everything has to be exactly right, or else CONSTRAINT_ERROR would raises its ugly head.

The difficulty of formatting the display on the screen almost lead me to fatal design error. This error is so common, and so important, it is worth while to devote the next subsection to it.

3.6.18. The Danger of Improvement

It's ironic, but sometimes you can improve a good product so much that it becomes useless. Several examples come quickly to mind. There was a word processing program that dominated the CP/M market in the late seventies. The manufacturer added many features to this good product, and released the new, improved version. The resulting product was so slow and difficult to use that it got terrible reviews in computer magazines, and other word processors tore the market away from it. There are two real-time operating systems that came out in the seventies that are suffering the same fate. Too many good products have failed because they've been improved too much. Some people can get upset and nasty when I criticize their products, so I'll pick on my own FORM_TERMINAL and show what almost happened to it.

The original FORM_TERMINAL consisted of six files that looked a lot like Listings 25 through 30. It did not have the capability of creating or editing forms. I used a text editor to create the external file containing the field specifications. As I pointed out, that was a nuisance, but it only had to be done once for each form I created, and I only created ten different kinds of forms. Each time I did it, it took less than an hour, so I spent less than 10 hours total time creating files with the text editor.

The FORM_TERMINAL is such a useful user interface, I wanted to be sure to include it in this book. I realized that its most serious deficiency was the laborious procedure required to create the form file. I decided to add the Create procedure, that would make this much easier. Well, after several days I got the Create procedure working, and it only increased the size of the FORM_TERMINAL package from six files to ten files. (Listings 31 through 34.)

I used the Create procedure for a while, and realized that it forced the user to start from scratch every time a new form was needed. If you wanted to correct an error in a form, or make a second form almost exactly like another form, you had to start from scratch. I needed a way to edit an existing form, so I wrote the Edit procedure.

The Edit procedure is spread out over eight files (Listings 35 through 42), and brought the total number of files in the FORM_TERMINAL to eighteen. Needless to say, this took considerable time and effort to get this feature working.

I discovered that while using the Create or Edit procedure it was possible to produce a form containing errors. I needed Error_Recovery to allow me to recursively call the Edit procedure until the form was error free. One more small file (Listing 43) brought the total to nineteen files in the FORM_TERMINAL package.

I used the Make_Form and Edit_Form programs (Listings 44 and 45) and discovered that the first call to Error_Recovery raises STORAGE_ERROR on my IBM PC AT clone. The publication deadline was getting close, and FORM_TERMINAL didn't work any more. I was panic stricken.

Finally I got the FORM_TERMINAL, as shown in Listings 25 through 43, to work on a VAX (and also on a PC if you don't make recursive errors). Most of your application programs won't use Create, Edit, or Error_Recovery. That means thirteen of the nineteen source files create dead code that will have to be removed by an optimizer (if you have one.)

Looking back, I see countless hours spent writing slick utility programs that save a few minutes. I was tempted to remove Create and Edit from the package specification, then remove all the code associated with them, and never tell you about them. That's less embarrassing to me, but I'd rather have you learn from my mistake. The whole sordid package is there for you to see.

It is easy to get seduced into doing more than you should. From time to time it is a good idea to ask yourself, "Is this really worth it?" Sometimes you have to admit you made a mistake and go back to an older version.

One easy way to return FORM_TERMINAL to its original small size is to use the Edit and Create stubs in Listings 46 and 47. If you compile these two small stubs, they write error messages if you should ever try to Edit or Create a form. I don't expect any of your application programs to call these routines, so they produce dead code, but not nearly as much as the real Edit and Create routines do. The better way, of course, is to edit the package specification and body to remove all references to Create, Edit, and Error_Recovery.

If I could live my life over again, I wouldn't have written the Edit and Create procedures; but the fact is that I did write them, and there are some lessons that can be drawn from them. Let's look at them.

3.6.19. Creating a New FORM

Create takes most of the work out of designing a form. You still need to decide what the form should look like, but the Create procedure does all the counting of lines and columns for you.

I wanted to make Create an independent program outside the FORM_TERMINAL package, but it needs to know about Field_specs and the SCREEN. I would have to make those internal details visible to all programs outside the package if the Create procedure was outside the package. I don't want clever application programmers directly manipulating the Field_specs and the SCREEN. Putting the Create procedure inside the package allows me to keep those details hidden from application programs.

The Create procedure asks you if you need instructions. If you do, it gives you a screen full of explanation. When you have read this, it covers the screen with '~' characters. The wiggles are there to help you see how much space you have to work with. (They won't appear on the form you create.) You can use the arrow keys to move the cursor around wherever you want, and type whatever you like. Keep doodling around until the form looks like you want it to. If you make a mistake, just type over what you have already done.

Eventually it will look like you want it to. When it does, it is time to tell the computer to store it. In general, you do this by pointing to the beginning and end of each field with the cursor and using function keys to indicate if it is protected or not. Each time you do this, the program will ask you to give the field a name. Every field must have a unique name, and must fit on a single line. Remember there is the concept of "next field" and "previous field", so be sure to specify them in the correct order (just as you must specify enumeration types in the correct order). Usually you will want to start with the field in the upper left corner and work to the right and down, but that's isn't necessary. (If you want to really baffle a user you can start at the bottom and work up!)

You point to the beginning of a field by moving the cursor to the first position in the field and press F1 or F2. Press F1 if this is to be a protected field the user can't modify. Press F2 if it is an area where the user is expected to enter data. Use the RIGHT arrow key to move the cursor to the last character in the field. (You can use the LEFT arrow key if you overshoot the end.) When the cursor is at the proper place, press F3. The computer stores the line number, the first and last column numbers, and the text contained in those columns. (The text could be a prompt, a default response, or blank spaces.) It also stores whether this field is protected or not. All that remains for you to do is to give it a unique name of 20 characters or less. You do this by typing the name at the prompt at the bottom of the screen and pressing the RETURN key. (Note: you may use significant embedded blanks and underlines, but all lower case letters will be converted to upper case automatically.)

After you have entered a field name the cursor returns to the end of the field you just entered, and you may enter the next field. When all the fields have been entered, press F10.

The Create procedure leaves the form in memory. You probably want to write it to a disk file. The Make_Form program (Listing 44) shows you how to do this. It doesn't do much more than call FORM_TERMINAL.Create and FORM_TERMINAL.Write.

3.6.20. Character Substitution

When I designed the SCROLL_TERMINAL I chose the question mark key as a help request. I couldn't imagine any time a user would answer a question with another question, so decided the question mark key should always raise the NEEDS_HELP exception. I kept the same convention in the FORM_TERMINAL. There were no problems until it came time to create a form.

It is likely someone will want a form to display a prompt with a question mark it in. How can someone create a form containing a question mark when pressing the question mark key always raises the NEEDS_HELP exception? The solution (near the end of the Process_Keystrokes procedure in Listing 33) was to substitute the escape key for the question mark. I don't like to map keys to other functions, but in this case it seemed like the best way to solve the problem.

3.6.21. Long Strings

Sometimes string literals won't fit on one line. Suppose you want to print a string that is 60 or 70 characters long. The print statement might appear at a point in the program where there are several levels of indentation, and you may be using dot notation, and your work processor may insist on saving a generous right margin. There isn't room to put SCROLL_TERMINAL.put_line("70 characters here"); on one line. The text editor inserts a carriage return somewhere in the string literal, and Ada generates an error saying something about an unterminated string.

I ran into that problem in Listing 32. The help messages wouldn't fit on a single line. The simple solution was to break the messages into two strings (one string on each of two lines) and print the catenation of the two strings. You can use this trick whenever a string literal won't fit on a single line.

3.6.22. IN OUT Mode

I am ashamed to say that, in my desperation to try to get the complete FORM_TERMINAL package to fit on a PC, I saved space by intentionally misusing the IN OUT mode in Listing 33. That was a really bad thing to do. Let me explain why.

Lazy programmers always use IN OUT mode to avoid those annoying error messages Ada generates when you misuse an IN or OUT mode parameter. Ada warns you of those errors for your own good. Using IN OUT mode to suppress them simply prevents you from detecting the error at compile time, and makes it appear at run time, when it is much more difficult to detect.

Pardon my FORTRAN, but Figure 30 shows what can happen if you don't pay attention to parameter modes. This is a fragment of a program I wrote for a client in FORTRAN because he didn't have an Ada compiler for his computer. FORTRAN treats all parameters the way Ada treats IN OUT mode parameters. TIME is expressed in milliseconds, and I wanted to convert it to HOURS, MINUTES, and SECONDS so I could display the time in "HH:MM:SS" format. The program did strange things because the value of TIME was corrupted by the SPLIT subroutine. For example, if the value of TIME was 34,644,822 before calling SPLIT, the display correctly showed 09:37:24, but the value of TIME was changed to 24,822. It took me most of a day to figure out what went wrong.

If I had written the routine in Ada, it would have looked like Figure 31. Since I thought I was just reading TIME and not changing its value, I would have declared it to have IN mode. Ada would have spotted my error at compile time. Then I would have rewritten it as shown in Figure 32. If I had used IN OUT mode for all the parameters in Figure 31, Ada would not have caught the error, and I would have had the same problem I had in FORTRAN.

```
Figure 30. Erroneous FORTRAN SPLIT subroutine.
----------------------------------------------------------
SUBROUTINE SPLIT(TIME,HOURS,MIN,SEC)
IMPLICIT NONE
INTEGER*4 TIME, HOURS, MIN
REAL*4 SEC

HOURS = TIME / 3600000
TIME  = MOD (TIME,3600000)
MIN   = TIME / 60000
TIME  = MOD (TIME,60000)
SEC   = TIME / 1000.0

RETURN
END
```

```
Figure 31. Erroneous Split procedure.
----------------------------------------------------------
with STANDARD_INTEGERS; use STANDARD_INTEGERS;
procedure Split(TIME        Integer_32;
                HOURS : out Integer_32;
                MIN   : out Integer_32;
                SEC   : out float) is
begin
  HOURS := TIME / 3600_000;
  TIME  := TIME mod 3600_000;    -- line 10
  MIN   := TIME / 60_000;
  TIME  := TIME mod 60_000;      -- line 12
  SEC   := float(TIME) / 1000.0;
end Split;

Meridian AdaVantage(tm) Compiler
[v2.1 Feb 29, 1988] Target 8086
"split.ada", 10: assignment to read-only object [LRM 6.2/5]
"split.ada", 12: assignment to read-only object [LRM 6.2/5]
15 lines compiled.
2 errors detected.
```

Notice the solution in Figure 32 requires the declaration of an extra variable. I didn't want to do that in Listing 33 because Form_specs take up lots of space. To be brutally honest, I was using DATA as a global variable, but pretending to pass it as a parameter. I should have made DATA an OUT parameter because DATA is produced by Get_Field. Then I should have declared a local variable of type Form_specs and copied it to DATA at the end of the procedure. (In fact, that's what I did in the original version. I had to take out the extra variable because it caused STORAGE_ERROR to be raised on the PC.)

Legitimate use of IN OUT mode is rare. It should only be used in those cases where you are passing a variable to a routine and you expect that routine to somehow modify it and return the modified value back to you. If you use IN OUT mode to avoid declaring an extra variable, your program may work, but it may confuse a maintenance programmer. He may spend hours trying to figure out where the calling program created the original value (it really didn't), or where the calling program will use the transformed value (it really doesn't). You shouldn't mislead someone into thinking a routine transforms a value if it simply uses it or produces it.

```
Figure 32. Correct Split procedure.
----------------------------------------------------------
with STANDARD_INTEGERS; use STANDARD_INTEGERS;
procedure Split(TIME        Integer_32;
                HOURS : out Integer_32;
                MIN   : out Integer_32;
                SEC   : out float) is
  T : Integer_32;
begin
  T     := TIME;
  HOURS := T / 3600_000;
  T     := T mod 3600_000;
  MIN   := T / 60_000;
  TIME  := T mod 60_000;
  SEC   := float(T) / 1000.0;
end Split;

Meridian AdaVantage(tm) Compiler
[v2.1 Feb 29, 1988] Target 8086
Subprogram body split added to library.
17 lines compiled.
No errors detected.
Meridian 8086 Code Generator
[v1.8 Jan 20, 1988] Target 8086 object
Generating code for split
```

3.6.23. Editing an Existing FORM

I sometimes became very frustrated with Create because I would almost be finished with a complicated form, and would make a little mistake. There was nothing I could do except start all over again. There was no way to edit the form.

If you have a complicated form with many fields, and you just want to add one more field, swap the position of two fields, or correct a spelling error in a prompt, you can't fix it with Create. Create will make you enter the entire form from scratch. That's a lot of unnecessary work, and it gives you too many chances to make a mistake.

The Edit procedure can be used to make changes to the Field_specs or SCREEN. The Edit_Form program, Listing 45, uses the Edit procedure to make it easy for you to make minor changes in the form.

3.6.24. null Exception Handlers

Students often make an amusing mistake they are first exposed to exceptions. They think they need to handle every exception in every routine. If they don't know what to do, the put a do-nothing exception handler at the end of the block.

```
begin
-- some code here
exception
     when others => null;
end;
```

I generally come down pretty hard on the student because he is telling Ada, "I don't know what went wrong, so just ignore it and proceed to the next block as if everything is OK." I used to say there is never a time when a null statement is an appropriate exception handler. Now I say it is ALMOST never appropriate. I used one in Listing 45.

The FORM_TERMINAL.Read procedure will raise LAYOUT_ERROR if it reads a form into memory from a file and then discovers an error in it. A LAYOUT_ERROR should be rare, and will probably force most programs to terminate abnormally. The Edit_Form program is a special case. When it reads a form from a disk there is a good chance there is a LAYOUT_ERROR in it. (That's why we want to edit it!) If we only allow the program to read good forms, then it isn't much use to us.

Notice that the Edit_Form procedure encloses the FORM_TERMINAL.Read(FILE); statement in a block and provides a local exception handler for that block. The exception handler ignores the LAYOUT_ERROR and lets the program proceed normally. Any other exception, like READ_ERROR, is not ignored and is handled by an exception handler at the end of the program.

3.7. Porting the IO Interface to VAX/VMS

Ada solves many portability problems, but there are always a few problems moving software from one system to another. These problems can be reduced if the program is written with portability in mind, but they can never be completely eliminated.

Almost all of the software in this book was developed on an IBM PC AT clone, using the Meridian Ada compiler. Moving to a genuine IBM PC AT with the Alsys compiler was no trouble at all. Most of the software was moved to a VAX running the DEC ada compiler under VMS without any modification. The only real trouble was moving some of the I/O routines to the VAX. That's not unusual. I/O typically causes portability problems.

At first it might seem surprising that some things that are easy to do on a microcomputer are hard to do on a minicomputer. A minicomputer is more powerful, but power does not always imply ease of use. A jack hammer is a powerful tool, but it is easier to use a less powerful 16 oz claw hammer for small jobs (like hanging a picture). The VAX is powerful, but for a small job, like a custom terminal interface, a smaller computer is easier to use.

There are two major differences between the PC and the VAX. The first difference is the number of users that are simultaneously supported. DOS on the IBM PC is a single-user operating system. This means it was designed to make it as easy as possible for the programmer to get direct access to system resources (disk files, terminal, printer, and so on). VMS is a multiuser system designed prevent users from interfering with each other. It does this by preventing direct access to system resources. The only way VMS lets you use system resources is through an operating system call which gives you limited access. When you are trying to directly control a peripheral, small single-user operating systems work for you, large multiuser operating systems work against you.

The second difference is that the DEC software is tightly coupled to DEC hardware. DEC terminals are not general purpose dumb terminals-- they have been specially designed to take some of the burden off DEC software. This specialization results in improved performance under normal circumstances, but lacks the flexibility needed for non-DEC applications.

Limited access provided by the operating system, and specialization of the hardware are typical problems encountered whenever software is ported from one system to another. This isn't a unique problem with VAX/VMS. It happens all the time. That's why it is important to try to hide I/O details in a package that provides a consistent virtual interface regardless of the underlying system. If you don't, you fight the same battle every time you port another application program.

3.7.1. VMS package

The VMS operating system doesn't want to encourage you to exchange individual character data with the terminal (it isn't as efficient as block transfers), so those system services don't exist. That's why I had to write a package similar to the Alsys DOS package, containing the three subprograms I needed. I called this package VMS. It is shown in Listings 48 through 51.

3.7.2. Raising Exceptions

The VMS package has to call some system services. These services return with a status variable telling if operation was successful. I don't know enough about the VMS operating system to know why the service request would fail, but I do know that I probably don't want to ignore the failure and try to proceed anyway. I need to do something about the failure, but I don't know what. In cases like these it is good idea to raise an exception and let the next higher level worry about it. (The technical term for this is, "passing the buck.")

If I raised a predefined exception, like CONSTRAINT_ERROR, that would really confuse someone if the error ever happened. I need to declare a user defined exception unique to this problem. I decided to call it VMS_IO_ERROR. That's not a very descriptive name. It doesn't tell what the problem is. It would be better to call it, TERMINAL_OFF_LINE, or something like that. I couldn't do that because I don't know what the problem is. All I know is that it is somehow related to a VMS I/O system service.

If I declared this exception in the package body, instead of the specification, then the VMS subprograms could still raise it. The exception would propagate out of the package as an anonymous (unnamed) exception because names inside a package body aren't visible outside the body. The only way the application program could handle this anonymous exception would be with a handler something like when others => Do_Something_Appropriate;. Since the application program doesn't know about it, it probably won't provide a handler for it, so the program will terminate with an unhandled exception if it is ever raised.

I was really tempted to do this, because I believe VMS_IO_ERROR has to be a fatal error resulting in the immediate termination of the program. I resisted the temptation because it isn't my place to decide the fate of someone else's application program. If I leave the exception buried in the body, a client will only see that the package VMS contains three subprograms and think that it doesn't raise any exceptions. What a nasty surprise when the program bombs with a cryptic error message like, "unnamed exception raised at PC = 00AF0166 never handled."

Placing the exception in the package specification warns the client that the exception can occur. The client can decide what to do about it. I doubt that there is much that can be done, because when VMS dies it cuts down on your options in a big way, but perhaps the client knows a clever solution. I don't want to deny the client the option to recover from the error.

3.7.3. CONTROL-C Powder Keg

Unfortunately VMS isn't as generous with options as I am. Whoever wrote the QIO service decided that nobody would ever want to pass CONTROL_C, CONTROL_Y, or CONTROL_Z back to a main program. Whenever a user presses one of these three control keys, the QIO service diverts the program flow to a VMS default handler which is reluctant to give control back to your program.

This is a serious problem on two counts. First, it forced me to pick a different character for the panic button. (I picked the exclamation point for no particular reason.) Now users have to remember the panic button is CONTROL_C on the PC, but it is the ! key on a VAX. I could solve this problem by changing the IBM PC versions so that the exclamation point is the panic character on those versions, too. That would make the user operation consistent regardless of the system, but it wouldn't solve the second part of the problem. Old timers like me are used to using CONTROL_C as a panic button. Whenever things run amok, we hit CONTROL_C by force of habit. The VMS intercept of the CONTROL_C leaves the user program running, and removes any possibility of the user regaining control.

There is a LIB$DISABLE_CTRL VMS service that might be used to disable CONTROL_C and CONTROL_Y, but it doesn't seem to do anything about CONTROL_Z. I tried to add it to the package body, but it got really messy. This isn't supposed to be a book about quirks in VMS, and these things don't really have much to do with Ada, so I decided to leave them out. If there is a VMS Wizard among the readers of this book, who can write an improved version of this package that isn't beyond the comprehension of mere mortals, I will be glad to include it in the next edition of this book. In the mean time, we just have to live with the danger.

3.7.4. Operating System Limitations

Situations like this one often lead to criticism of Ada. The charge is that Ada doesn't have enough low-level capability, or Ada's run-time system is inadequate. This isn't an Ada problem, it is an operating system problem. The CONTROL_C problem doesn't exist on DOS, and I would have the same problem on the VAX if I were writing this FORTRAN, C, or assembly. The fact that my Ada program doesn't handle CONTROL_C on VAX/VMS isn't because Ada is inadequate-- it's a VAX/VMS limitation. If you can show me how to get unfiltered characters from the keyboard in FORTRAN running under VMS, I bet I can show you how to do it in Ada using the same technique. Ada just makes operating system limitations more visible because Ada programs attempt more ambitions projects. (Can you imagine trying to write the FORM_TERMINAL in FORTRAN?)

3.7.5. INPUT task

Listing 49 shows the VMS package body. The input subprogram bodies simply call entries in a task called INPUT. A complete treatment of tasks is beyond the scope of this book, so I was hoping to avoid the subject completely, but VMS forced me to use this one. Here is a quick overview of this particular task.

The INPUT task (Listing 50) has three entries: Keypush, Ready, and Get. It is generally suspended, waiting for one of those three entries to be called. Keypush is an asynchronous system trap (AST) entry. It is asynchronous because it happens whenever a user presses a key. (That is, it doesn't always happen at a certain point in the program.) It tells the INPUT task that it must process the input character so the user can press another key. You can consider it to be the input port of a buffer between the asynchronous input from the user and the synchronous requests for data from the application program. The Get entry point is the output port of the buffer. It waits to supply data to the program until the program asks for it, or makes the program wait until data is available. (It synchronizes the data with the program.) The Ready entry point tells the application program if there is any unprocessed data in the buffer or not. This is useful in programs that can be doing other things while waiting for data. The program can poll the Ready entry point periodically and process input data (if it is there) at its convenience.

The INPUT task is filled with "secret sauce" unique to VMS. It would have taken me forever to figure this out myself, but fortunately Lee Lucas and Dave Dent had to solve this problem before I did, and I was able to take advantage of their work. They didn't come up with this all by themselves. They adapted an earlier program by Dee DeCristofaro for their use. I say this not only to give credit where credit is due (and shift blame away from myself if this is a dumb way to do it), but also to make a software engineering point. Even in those cases where software can't be reused without change, modular programming can make it easier to adapt software from one application to another. Lee and Dave structured their software in such a way that I was able to easily recognize the parts I needed and could extract them for my use.

DEC Ada comes with two unique packages for interfacing with VMS system services. The main one is STARLET. (The name simply means that some of the DEC programmers like to name software modules after constellations, stars, planets, and so on.) If you want to print a copy of the STARLET package specification, be sure you dump it to a high speed line printer with a nearly full box of paper. I might have named the package NOT_KITCHEN_SINK because that is one of the few things it doesn't contain. Despite all that, it doesn't include the definition of Cond_value_type, so a second package, CONDITION_HANDLING, is needed too.

Finally, a there is a package called SYSTEM which gives the definition of address types, and other system dependent declarations. Section 13.7 of the LRM allows some special features to be added to this package. The DEC version includes conversions of addresses and integers to unsigned long words, and a function Or, which acts as a bit set operation.

The task body INPUT just glues pieces of these DEC- specific packages together. The task begins by calling STARLET.Assign to assign the user's terminal, called SYS$COMMAND, to a CHANNEL. When it does this it assigns a value of STARLET.Channel_type to the local variable CHANNEL. It also assigns a value of CONDITION_HANDLING.Cond_value_type to a variable called ASG_STATUS. A boolean function called CONDITION_HANDLING.Success knows how to examine the ASG_STATUS and decide if the operation was successful or not. The Assign operation should always succeed, so the Success function should always return TRUE when it examines ASG_STATUS. If it doesn't, the INPUT task raises VMS_IO_ERROR and gives up. (I've never seen that exception raised, and hope I never will.) Once the channel is assigned, it is ready for use. It need not be assigned again.

It may bother you that we don't know what the value of CHANNEL or ASG_STATUS is. It shouldn't. We don't need to know if these are integers, strings, or enumeration types. Keeping this information hidden from us prevents it from distracting us. If we knew CHANNEL was an integer, we might get lazy and just assign it the value of 27 instead of using SYS$COMMAND. That might work when we logged on at our usual terminal, but not when we logged on at another one.

After the terminal is assigned to a channel, the task enters a loop that tells the VMS operating system to get a keystroke from the keyboard, waits for keystroke, and then gives it to the client program. This loop continues as long as the VMS package is in scope. Since the VMS package is normally WITHed by VIRTUAL_TERMINAL, which is WITHed into a package like SCROLL_TERMINAL, which is WITHed into the main program, the loop continues until the main program ends.

When the loop begins, no data has been received yet. Therefore, the variable NEW_DATA is set to FALSE. This fact will be used to guard an entry point later in the task.

The STARLET procedure Qio starts an I/O operation and returns immediately without waiting for completion. The operation uses the CHANNEL that was assigned to SYS$COMMAND, and the function to be performed is to read a virtual block. This function is modified by IO_M_NOECHO, which tells it not to echo the characters to the screen as they are received. Furthermore, the IO_M_NOFILTR tells it not to interpret CONTROL_R, CONTROL_U, or DELETE, as editing characters, and passes them along to the task. (Alas, it still filters CONTROL_C, CONTROL_Y, and CONTROL_Z as we have already mentioned.)

The success of the Qio procedure is stored in QIO_STATUS, and it is interpreted by CONDITION_HANDLING.Success just as the ASG_STATUS was. The status of the operation (presumably values like ready, pending, in progress, complete, transfer count) is stored in QIO_IOSB, which is of type STARLET.IOSB_type. QIO_IOSB will have a transfer count of 0 the first time it is read, but later, after the user has pressed a key, the transfer count will be 1. The value of this variable will change as the result of a direct memory access operation, or perhaps as the result of an interrupt service routine.

The pragma Volatile(QIO_IOSB) tells the optimizer that QIO_IOSB can change without program intervention. Without that pragma, an optimizer might realize that it had already read QIO_IOSB and saved the value in a register. It would keep rereading the register and always find the same result.

KEYINPUT is a string, where the Qio procedure will put the input data. Normally this string is several characters long for efficiency, but I want to process each character individually, so I made the string one character long. The parameters P1 and P2 tell the Qio procedure where the string is and how long it is.

The ASTADR parameter in Qio tells the procedure the address of the Keypush entry. When the user presses a key, Qio will transfer the value of the key to KEYINPUT(1). Then Qio calls the Keypush entry to let the application program know there is new data in KEYINPUT(1).

After the Qio operation has been successfully initiated, the task enters an inner loop. This inner loop allows three alternatives. It can (1) accept a Keypush, (2) report that no keys have been pressed yet, or (3) terminate. The second alternative can happen multiple times. The first alternative can happen only once because it sets NEW_DATA to TRUE, which exits the inner loop. The last alternative, terminate, can happen only once.

Eventually the user will press a key, and the task will enter a second inner loop. It is strikingly similar to the first inner loop, except its first alternative is to accept a Get instead of a Keypush. The second inner loop ends when the Get entry is called. The Get entry copies the input character to the OUT parameter and sets NEW_DATA to FALSE to exit the loop. The outer loop calls the Qio procedure again and the cycle repeats.

Most of the time, the INPUT task is suspended, waiting for something to happen. When the main program ends, it will be sitting on a select statement that includes a terminate alternative, so it will end when the main program ends.

3.7.6. OUTPUT package

The OUTPUT package (Listing 51) looks a lot like the INPUT task. In fact, originally the source file was created by editing the INPUT task source code. Let's look at the differences.

The obvious difference is that it is a package, not a task. Since the first version was derived from a copy of INPUT, it used a second task for OUTPUT and it worked just fine. I could have left it a task, but I elected to change it to a package.

I think it is better not to use a task, not because of the task switching overhead, but mostly for a philosophical reason. Tasks should be reserved for independent, concurrent activities. The INPUT task deserves to be a task because it can be considered to be a separate program continuously scanning the keyboard so the main program doesn't have to.

If I were sharing a printer with other users, it would make sense to make OUTPUT a task that buffers output characters, checks for printer availability, and sends the characters when the printer is ready. That would allow the main program to continue processing even though the printer isn't available. That isn't the case here. I'm sending this data to the user's terminal, which should always be available. Generally I'm sending a prompt and waiting for the user's response. There is no sense running ahead to look for a response before the prompt is sent. So OUTPUT shouldn't be a separate thread of control. It is just another step in a sequential process.

That's why the Put procedure uses STARLET.Qiow instead of STARLET.Qio. The w stands for wait. I don't want to run ahead until the output character is on its way to the user. I have to wait until the character is sent, so I might as well give up the processor and let someone else use it.

3.7.7. Enforcing Order

Why isn't Put just a procedure? Why did I stick it in a package? The key is in the last few lines of the package. The output channel has to be assigned before data can be sent to it.

This raises a portability issue. I want to port all my programs developed under DOS to VMS, and none of my existing application programs assign the output channel because it isn't necessary on DOS. If I didn't hide this channel assignment in the elaboration of some package, I'd have to change all my application programs to port them to VMS.

Even if there wasn't a portability problem, there would still be a compelling reason to stick Put in a package. There is a procedural order that must be enforced: First assign the output channel once, then use it many times. If I didn't use a package, I would have to depend on every application programmer that ever uses the VMS package to remember to assign the output channel before use. If the application programmer forgot to do that, who knows what error would happen. I have to make sure the procedures are called in the correct order.

When an application program running under VMS uses SCROLL_TERMINAL it WITHs in VIRTUAL_TERMINAL, which elaborates the package VMS, which elaborates OUTPUT, which runs STARLET.Assign. All this happens automatically so the application program doesn't have to remember to assign the output channel. Furthermore, it all happens before the application program gets control, so the application program can't write to the output before it is assigned.

Tasks can also be used to enforce order. The INPUT task assigned the channel first and then used select statements to assure that Get was not accepted until after a key was pressed.

I think failure to consider order of execution is a major problem in Ada programming. I suspect this is because most programmers are used to programming single-task programs that automatically enforce order because they have a single thread of control, so they aren't used to thinking about it. Programmers who are used to writing programs with multiple tasks are more likely to think about it, but they may be tempted to rely on secret things they have discovered about their operating system scheduling algorithm, because that's the way they've always done it. Their programs aren't likely to be portable.

The Ada language does contain features that enforce an order of execution. The INPUT task and OUTPUT package are examples of how to use these features. Use them wisely.

3.7.8. Hardware Limitations

There is also a hardware interface problem on the VAX. If you are using a VAX you are almost certainly using a VT52, VT100, VT220, VT240, or something that emulates a DEC terminal. When you look at all those extra keys on the right side of a VT100, and the row of function keys across the top of the VT220, you would think there would be no problem making it compatible with the VIRTUAL_TERMINAL. Although a DEC terminal appears to have all the capability you need, it doesn't. Many of those keys don't get past the keyboard. For example, function keys F1 through F5 on a VT240 are dedicated to Hold Screen, Print Screen, Set-Up, Data/Talk, and Break. Pressing these keys makes the terminal do something, but sends nothing to the computer, so no amount of clever software can process them. The INSERT and DELETE keys don't exist on a DEC terminal, either.

The situation is a little better if you use a dumb terminal instead of a DEC terminal. The Televideo 910, for example, has fewer keys than a DEC terminal, but most of them produce unique codes. Even so, there are still a few problems. The DOWN arrow key on a TV910 produces the same ASCII code as the LINE_FEED key. Therefore, you really don't have a DOWN arrow key, you just have two LINE_FEED keys with different legends on the key cap, and there is no way for software to tell them apart.

The possibility of missing keys is the reason the VIRTUAL_TERMINAL package specification defines special control codes. A terminal may not have an INSERT key, but it will certainly have a control key and an A key, so the user can press CONTROL-A to simulate the INSERT key to add text. Even if the terminal doesn't have a DELETE key, the user can press CONTROL-E to erase characters. The VIRTUAL_TERMINAL package will work on any terminal, but it may be awkward on some terminals because they don't have the required keys.

The FORM_TERMINAL.Create procedure is more awkward to use on a DEC terminal than an IBM PC because function keys F1, F2, and F3 don't exist on the DEC terminal. FORM_TERMINAL.Create uses F1 to mark the beginning of a protected field, F2 to mark the beginning of an unprotected field, and F3 to mark the end of either. When creating a form on a DEC terminal you have to use CONTROL_F followed by a 1, 2, or 3 to simulate those missing keys. The program works, but it isn't as convenient as it would be if those function keys existed.

If you port the virtual terminal to your physical terminal, you may have some unused keys that produce unique codes. You may want to define these keys to replace missing keys. Just modify the VIRTUAL_TERMINAL package body so the unused keys get mapped to the control codes specified in the package specification. (If you do this, it might be confusing if you ever change terminals, but that's a price you might want to pay for convenience.)

3.7.9. DEC VIRTUAL_TERMINAL

Despite all these problems, it is possible to port the VIRTUAL_TERMINAL package to DEC Ada. The DEC version is shown in Listings 52 and 53. In theory, I shouldn't' have to change the package specification, but I did because of the way VAX/VMS handles control characters, and because my VMS package may want to raise an exception that the PC version doesn't need to raise. Since I wrote the VMS package to look a lot like the Alsys DOS package, the VIRTUAL_TERMINAL package body for VAX/VMS is almost identical to the Alsys body.

All I/O specific code is confined to the VIRTUAL_TERMINAL package. The more powerful SCROLL_TERMINAL and FORM_TERMINAL packages are built on top of the VIRTUAL_TERMINAL package. Therefore, once the VIRTUAL_TERMINAL is running on VAX/VMS, the SCROLL_TERMINAL and FORM_TERMINAL can be ported to VAX/VMS without modification.

3.8. VIRTUAL_PRINTER package

If you have ever tried to use TEXT_IO to send data to a printer, you know you have to know the name of the printer to open it. If you are using DOS on a PC, there is a good chance the printer will be connected to LPT1, but it doesn't have to be. On VMS the printer could be on just about any port, so you can't assume TXA7: will always be the printer port. (In most cases it won't.)

3.8.1. What's Its Name?

What are you going to do if you want to write a program that sends data to a printer? You could ask the user the name of the printer every time. That's a bad idea because it is a nuisance to the user, and the user might not even know the name of the printer port. You could code the printer port name right into your application program, but that can cause maintenance and portability problems. Suppose the central computing facility reconfigures the system, and moves the printer to another port. You would have to search through all your application programs to find every place you have named the printer port. You would have to do the same thing if you wanted to use your programs on another computer.

My solution is to write a package called VIRTUAL_PRINTER (Listing 54) which has the name of the printer hidden in the body. If you ever need to change the name of the printer port, you just have to change the name in the VIRTUAL_PRINTER body, recompile it, and relink any application programs that use it. You don't need to recompile the affected application programs because they depend on the VIRTUAL_PRINTER specification, and you've only had to change the body. All you have to do is relink them. (An Ada programming environment ought to be able to give you a list of all units that depend on the VIRTUAL_PRINTER package specification, so you will know which programs to relink.)

3.8.2. Printer Quirks

The VIRTUAL_PRINTER body is also a good place to take care of printer quirks. Many years ago I ran across a printer that used an escape sequence instead of the normal ASCII form feed character to advance to the top of a page. There may not be any of those printers around any more, but if there are, you can put some code in the VIRTUAL_PRINTER body that substitutes the appropriate escape sequence whenever it receives a form feed.

Figure 33. VIRTUAL_PRINTER body for COM2 port.

```
-----------------------------------------------------------
--             VPBCOM2.ada - 20 October 1987 - Version 2.0

--             Do-While Jones - 324 Traci Lane - Ridgecrest, CA 93555
--             (619) 375-4607

-- This version works for Alsys & Meridian Ada on an IBM
-- with the printer connected to the COM2 serial output port.

-- The printer connected to COM2 is an ancient Microline 83,
-- which seems to be a little slow to tell the PC that it
-- can't accept another character because it is printing a
-- line. Consequently, the first character of a line
-- sometimes gets lost. The solution is to send an ASCII.NUL
-- after every CR and LF. If the NUL gets lost, no harm is done

with TEXT_IO;
package body VIRTUAL_PRINTER is

  PRINTER : TEXT_IO.File_type;

  PRINTER_NAME : string(1..80);
  LENGTH       : natural;

  procedure put(C : character) is
  begin
    TEXT_IO.put(PRINTER,C);
    -- the following is required by the Microline 83 printer
    if C = ASCII.CR or C = ASCII.LF then
      TEXT_IO.put(PRINTER,ASCII.NUL);
    end if;
  end put;

  procedure put(S : string) is
  begin
    for i in S'RANGE loop
      put(S(i));
    end loop;
  end put;

  procedure Set_Output(DEVICE : string) is
  begin
    TEXT_IO.Close(PRINTER);
    TEXT_IO.Create(PRINTER,TEXT_IO.OUT_FILE,DEVICE);
    LENGTH := DEVICE'LENGTH;
    PRINTER_NAME(1..LENGTH) := DEVICE;
  exception
    when TEXT_IO.NAME_ERROR =>
      raise NAME_ERROR;
    when TEXT_IO.USE_ERROR =>
      raise USE_ERROR;
  end Set_Output;

  function Current_Output return string is
  begin
    return PRINTER_NAME(1..LENGTH);
  end Current_Output;

  function Standard_Output return string is
  begin
    return "COM2";
  end Standard_Output;

begin
  TEXT_IO.Create(PRINTER,TEXT_IO.OUT_FILE,Standard_Output);
  LENGTH := 4;
  PRINTER_NAME(1..LENGTH) := Standard_Output;
end VIRTUAL_PRINTER;
```

The first printer I attached to my IBM PC AT clone was my old reliable Microline 83 printer, which had served my CP/M system for five years. It was configured for 1200 baud serial operation, and I put it on the COM2 port. Modern printers buffer several thousand characters, but in 1982 serial printers only buffered a line or two. At the end of each line the Microline 83 used an RS-232C control signal to tell the computer not to send any more data because it was busy printing the data in the buffer. My new computer expects the printer to use CONTROL_Q and CONTROL_S to tell it when to start and stop sending data, and must not bother to check the RS-232C control lines. Consequently, the first character of a line would sometimes be sent while the printer was still printing, and wouldn't get into the print buffer.

This is not a new problem. The original teletypewriter terminals took a long time for the print cylinder to return to the first column. They used a 20 milliamp current loop, and didn't have any extra control lines in those days. Since they didn't have any way of telling when the print cylinder was ready to print again, they routinely sent a few null characters after every carriage return. Nulls aren't printed, so it doesn't matter if they get lost or not. (Now you know why electronic bulletin boards sometimes ask you if you require any nulls.) Figure 33 shows that I solved my problem in exactly the same way.

When I bought my Epson LQ-850, I put it on the parallel port LPT1. All I had to do to was make the changes shown in Listing 55, and all my old programs would send data to by new printer.

You will almost certainly have to write your own VIRTUAL_PRINTER body, but it shouldn't be too difficult. All you have to do is know the logical name of the printer.

3.9. SCROLL_PRINTER package

Just as the VIRTUAL_TERMINAL provided a portable base for the SCROLL_TERMINAL, the VIRTUAL_PRINTER is the base for the SCROLL_PRINTER. The SCROLL_PRINTER specification is shown in Listing 56, and the body is in Listing 57. It's basically just the SCROLL_TERMINAL with the input parts removed, so there isn't any more that needs to be said about it.

Chapter 4. PROGRAMMING ISN'T SOFTWARE ENGINEERING

The difference between programming and software engineering is like the difference between gardening and farming. You could say the difference is the size of the effort, but there is really more to it than that.

Farming isn't just gardening on a large scale. You can't use the same techniques for farming that you would if you were gardening. Any farmer who tries to plant his crops using nothing more than a shovel, rake, and hoe, is not going to succeed. Farming requires more powerful tools. A farmer needs a tractor.

Gardening isn't just farming on a small scale. You can't use the same techniques gardening that you would use if you were farming. I shouldn't try to use a tractor to plant my six tomatoes seedlings. It would be more trouble just getting the tractor into my back yard than it would be dig six holes with a shovel. The amount of money I save growing my own tomatoes wouldn't pay the maintenance on the tractor.

Even though there are differences between gardening and farming, there are some fundamental principles that don't change. Regardless of the size of the effort, you still need to provide the plants with adequate nourishment, water, and the right amount of sunlight. Things that you learn about soil preparation will be useful to you regardless of whether you are gardening or farming.

Software engineering isn't just programming done by more people over a longer period of time. You need different techniques for "programming in the small" and "programming in the large." In this section you will see several examples of small programming projects and one example of software engineering. Some of the techniques that work for small programming projects aren't adequate for large projects. Some of the techniques necessary for large projects are too awkward for small projects. Some basic principles (like the ones discussed in the previous sections on numeric considerations and IO utilities) hold for both programming and software engineering.

I'm sure you wouldn't try to plant a 40 acre farm with just a shovel, nor would you be foolish enough to try to use a tractor to plow a 5 x 10 foot backyard garden. Most people intuitively know when an area of land is too big to shovel or too small to plow. Unfortunately many people lack that same intuition when it comes to software development. They have one method, and they use it regardless of the size of the project. Using software engineering techniques on a small program leads to just as much trouble as using simple programming techniques on a large project does. You will save yourself a lot of grief if you can recognize when to shovel and when to plow.

You are about to see several little software tools that are examples of programming. They took a few hours to write and debug. I didn't spend weeks planning them; I just started writing with a vague goal in mind. As I got closer to the goal my vision became clearer. I used the programs and then made minor modifications to improve them. That is an appropriate approach to take for small projects.

If the project is large, seat-of-the-pants programming just won't work. You don't just sit down one afternoon and write the operational flight program for the space shuttle. You can't just say to yourself, "I'm not really sure what this space shuttle software should do, but it will come to me if I just wing it." Big programs require software engineering. The Draw_Poker program is a small example of a big program, and it shows some of the things you have to do differently when working on a large project.

I wish I could give you a simple rule, like, "Use simple programming techniques for projects less than 1,000 lines of code, and use software engineering for larger projects.", but I can't. There isn't a clear cut boundary between big and small that can be expressed in lines of code. Even if there were, it wouldn't do you any good because you don't know how many lines of code there are in the program until it is finished, and then it's too late.

Still, there are ways to tell when a project warrants software engineering. Ask yourself, "Is this program likely to require long-term maintenance? Will there be people on salary who will be responsible for improving this program and correcting bugs? Is this a program that will take several man-years to develop?" If the answer to these questions is yes, then you should use software-engineering principles. If not, applying rigorous software-engineering discipline will simply make a small project cost as much as a large one, with little or no benefit.

4.1. The Show Tool

I was doing a job for a client, using his Alsys Ada compiler on his IBM PC AT. It was my first experience with Alsys Ada and my first experience with PC DOS. I discovered that when you compile SOMEFILE.ADA, and SOMEFILE.ADA has errors in it, Alsys Ada will write the errors to SOMEFILE.LST without displaying them on the screen. Alsys gives such complete error messages that every error gives you at least five lines of text describing the error and suggestions of how to correct it, so SOMEFILE.LST can easily contain two or three screens full of error messages.

Since I was inexperienced with PC-DOS, I tried to display SOMEFILE.LST the same way I would on VMS. I used the command TYPE SOMEFILE.LST. The DOS TYPE command does not pause at when the screen fills, and text written to the screen isn't limited by a 1200 baud modem, so the whole error file flashed across the screen in a blur. I had to use CONTROL-S and CONTROL-Q to interrupt the transmission of text to the screen, and I had to have very fast reflexes.

I've had some limited experience with UNIX, so I tried MORE SOMEFILE.LST. That appeared to crash PC DOS, so I had to reboot. (It didn't really crash, but the symptoms were the same as a crash. The explanation of what really happened is best delayed for a little while.)

In desperation I read the PC-DOS documentation. The section on the TYPE command told me there were no switches I could set to display one screen at a time, and there was no cross reference to the MORE command.

I knew it was easy to write a program to display one screen of data at a time, because I had done it years ago in 8080 assembly language. I decided to rewrite it in Ada. I called the first version of that program Show, and you can find the listing for it in Figure 34. It prompts the user for the file name, opens the file, then does a loop 22 times that gets one line from the file and writes that one line to the screen. It prompts the user to "Press RETURN for the next screen," and jumps back to the loop that copies 22 lines to the screen. It does this until it hits an end of file mark.

4.1.1. Named Loops

Let's digress for a moment, and talk about named loops. I will sometimes use a comment to describe what a loop is doing, but I don't name a loop unless I'm going to use an unusual exit from the loop. The label MAIN: in Figure 34 should draw attention to that loop.

The problem is that I have nested loops. The inner loop executes 22 times, and the outer loop executes as often as necessary, until there is no more data to display. I could begin the outer loop with while not End_Of_File(FILE) loop if I were certain every file would have an exact multiple of 22 lines in it. In general, that won't be true. The end of the file will almost always be reached after a partial screen has been displayed. If I just write exit when End_Of_File(FILE);, that will get me out of the inner loop, but not out of the outer loop. The program would prompt the user to press RETURN, then go to the top of the outer loop again, where I would have to check for the end of file again.

The chicken way out is to never check end of file, let the program run until it raises an exception, and quit in the exception handler. That works, but it requires some intuition on the part of a maintenance

Figure 34. The first version of Show.

```
---------------------------------------------------------
--             SHOW.ada
--             9 June 1987
--             Do-While Jones

with TEXT_IO; use TEXT_IO;
procedure Show is
  TEXT   : string(1..200);
  LENGTH : natural;
  FILE   : File_type;
begin
  put("What file? ");
  get_line(TEXT,LENGTH);
  Open(FILE, IN_FILE, TEXT(1..LENGTH));
  MAIN:
  loop
    for i in 1..22 loop
      exit MAIN when End_Of_File(FILE);
      get_line(FILE, TEXT, LENGTH);
      put_line(TEXT(1..LENGTH));
    end loop;
    put_line("Press RETURN for the next screen");
    get_line(TEXT,LENGTH);
  end loop MAIN;
exception
  when NAME_ERROR =>
    put("File ");put(TEXT(1..LENGTH));
    put_line(" could not be found.");
end Show;
```

programmer to figure out how the program ends (unless you reveal the trick in a comment). I find that solution artistically offensive because it looks sloppy and careless. Besides, exception handlers should be used for unusual error conditions. A finite length file is not unusual or erroneous.

The assembly language version handled the problem by checking for end of file at the point corresponding to the exit statement. If it found it was at the end of the file, it jumped to a statement corresponding to Close(FILE);. (JUMP is the assembly language equivalent of GOTO.) I certainly didn't want to endure the shame of using a GOTO in an Ada program, so I didn't use that solution, either.

Naming the outer loop is the clean solution to the problem. The exit MAIN when End_Of_File(FILE); statement takes us down to end loop MAIN; as soon as we run out of data. It clearly shows the maintenance programmer the condition required to leave the loop, and is considered a normal exit.

4.1.2. Command Tail

Now let's return to the story of the development of the Show tool. I wasn't happy with the program because it still wasn't as good as the assembly language version. When I used the Ada version I had to wait for the program to prompt me for the file name. The assembly language version let me type a single command, SHOW SOMEFILE.LST, and automatically extracted the file name SOMEFILE.LST from the command line without me having to enter it in response to the prompt.

The Ada LRM doesn't specify a standard way to get the rest of the command line. In all fairness to Ada I should point out that most other languages don't do that either, because it is beyond the normal scope of the language. It is really an operating system function.

I looked through the Alsys documentation and found a package called DOS that includes a function Get_Parms that fetches the command tail for you. It wasn't exactly what I wanted, because I wanted something that was an exact replacement for the get_line procedure already used in the Show program. I knew I would have portability problems if I used Get_Parms in Show and tried to move Show to another system, because Get_Parms is a special function Alsys was thoughtful enough to provide with their compiler. It almost certainly wouldn't exist in any other Ada implementation.

4.1.3. Compiling Library Procedures

I decided the best approach would be to write the procedure Get_Command_Line, shown in Figure 35. It produces a string and a length, just like get_line. All the implementation specific code is confined to one place, and does not infect all the application programs that need to read the command tail. Once this procedure is compiled and stored in the Ada library, every application program that needs to read the command line can use it. Novice Ada programmers think that only packages can be compiled and reused as library components. That's not true. This is an example of a procedure that can be compiled once and reused often.

4.1.4. Unconstrained Strings

Implementation specific routines are usually tricky, and Figure 35 is no exception. The Get_Parms function returns an unconstrained string L characters long, where L depends on the number of characters typed after the command name. I want to write LENGTH := L; TAIL(1..L) := DOS.Get_Parms;. The problem is, I don't know what L is. The incredibly clever solution is to make the function call an input parameter to the Extract function. Extract(DOS.Get_Parms, TAIL, LENGTH); associates the string returned by DOS.Get_Parms with the formal parameter S_IN. The LENGTH attribute yields the value of L. Notice that I couldn't write S_OUT(1..L) := S_IN; because L is an out parameter and can't be read. That's not a problem because I can use the LENGTH attribute as often as I want.

Figure 35. The first Get_Command_Line procedure.

```
-------------------------------------------------------
--             GCLBAIBM.ada
--             19 October 1987

--             Do-While Jones
--             324 Traci Lane
--             Ridgecrest, CA 93555
--             (619) 375-4607

-- This version works with Alsys Ada on the IBM
-- PC. The Alsys DOS package contains a
-- function Get_Parms which returns a string
-- with a length depending upon the number of
-- characters entered by the user. Since this
-- probably isn't the exact number of
-- characters requested by the calling program,
-- a little bit of data massaging has to be
-- done to put the command line in the first
-- part of the longer output string.

with DOS;
procedure Get_Command_Line(S : out string;
                           L : out natural) is

  procedure Extract(S_IN :     string;
                   S_OUT : out string;
                       L : out natural) is
  begin
    L := S_IN'LENGTH;
    S_OUT(1..S_IN'LENGTH) := S_IN;
  end Extract;

begin
  Extract(DOS.Get_Parms, S, L);
end Get_Command_Line;
```

I thought that trick was pretty clever, until I read a column by Ben Brosgol. He knew an even easier way to use the Alsys DOS.Get_Parms function. (He has an unfair advantage over me. He works for Alsys!) When you declare a constant string, you don't need to declare the bounds because the bounds can be determined from the value you assign to the constant, even if that value is an unconstrained string returned by a function. I incorporated his idea into my procedure and came up with Listing 58.

4.1.5. Using Library Procedures

The Show program was compiled in the context of the Get_Command_Line procedure, as shown in Figure 36. TEXT and LENGTH come from Get_Command_Line if the user enters a file on the command line, or from get_line if he doesn't. (I like the way VMS prompts for missing parameters, so I usually include that feature in my routines.)

4.1.6. Porting Show to Other Systems

It wasn't long until I bought an AT clone of my own, and Meridian Adavantage Version 1.5 compiler to go with it. I decided to port the Show program to my computer. As I expected, Meridian had a utility that could fetch command line arguments, but it didn't look anything like the Alsys Get_Parms function. "No problem," I thought, "I'll just enclosed it in a different version of Get_Command_Line."

My first attempt at doing this must have looked a lot like Figure 37. I compiled Get_Command_Line and then compiled Show, linked them, and tried the resulting executable code. There was something wrong with it. The main program Show was fully debugged, so the error, of course, was in Get_Command_Line. I thought I knew what was wrong with Get_Command_Line, changed it, and recompiled it. I tried to relink it with Show, but the compiler (correctly) told me that Show was obsolete. Show was compiled in the context of Get_Command_Line, and since I had changed Get_Command_Line Ada couldn't be sure

Figure 36. Improved version of Show.

```
--------------------------------------------------------
--             SHOW.ada
--             9 June 1987
--             Do-While Jones

with Get_Command_Line;
with TEXT_IO; use TEXT_IO;
procedure Show is
  TEXT   : string(1..200);
  LENGTH : natural;
  FILE   : File_type;
begin
  Get_Command_Line(TEXT,LENGTH);
  if LENGTH = 0 then
    put("What file? ");
    get_line(TEXT,LENGTH);
  end if;
  Open(FILE, IN_FILE, TEXT(1..LENGTH));
  MAIN:
  loop
    for i in 1..22 loop
      exit MAIN when End_Of_File(FILE);
      get_line(FILE, TEXT, LENGTH);
      put_line(TEXT(1..LENGTH));
    end loop;
    put_line("Press RETURN for the next screen");
    get_line(TEXT,LENGTH);
  end loop MAIN;
exception
  when NAME_ERROR =>
    put("File ");put(TEXT(1..LENGTH));
    put_line(" could not be found.");
end Show;
```

Figure 37. The original Get_Command_Line body for Meridian.

```
-------------------------------------------------------
--            GCLBMIBM.ada
--            9 June 1987
--            Do-While Jones

--  Meridian command line interface for IBM PC.

--  This procedure is NOT PORTABLE because it uses
--  some utility packages available from Meridian
--  Software Systems, Inc. These packages are
--  compatible with the Meridian AdaVantage compiler,
--  but are not included with the compiler. (They
--  must be purchased separately.)

with TEXT_HANDLER, ARG; -- Meridian Utility packages.
procedure Get_Command_Line(TAIL   : out string;
                           LENGTH : out natural) is
  BUFFER : TEXT_HANDLER.Text(127);
  LEN    : natural;
begin
  if ARG.Count < 2 then
    LENGTH := 0;
    return;
  end if;
  TEXT_HANDLER.Set(BUFFER, ARG.Data(2));
  LEN := TEXT_HANDLER.Length(BUFFER);
  LENGTH := LEN;
  TAIL(1..LEN) := TEXT_HANDLER.Value(BUFFER);
end Get_Command_Line;
```

Show was still valid. I recompiled Show, linked, ran the program, and it still didn't work. I modified Get_Command_Line again, recompiled it, then had to recompile Show again, and so on. Eventually I came up with the form you see in Figure 37, but it was frustrating having to recompile Show every time.

It happened that I was evaluating the Gould APLEX Ada compiler running under the MPX-32 operating system at the time. I decided to try to transport the Show program to it. I was not very well acquainted with MPX-32 or APLEX Ada, and I knew I was going to have to recompile Get_Command_Line a million times before I got it working. That didn't bother me. I expected that. What bothered me was that I was going to have to recompile Show every time, even though I knew it was correct and never changed it.

I wished I had put Get_Command_Line inside a package. Then I could have compiled the package specification, compiled Show, and compiled the package body containing Get_Command_Line last. Then I could recompile Get_Command_Line as often as it took to get it working, and Show wouldn't need to be recompiled because it wouldn't be obsolete. (Show would depend upon the package specification, not the body.) Then I realized the value of a widely ignored Ada feature. You can separately compile a procedure specification.

4.1.6.1. Compiling Procedure Specifications.

Ada programmers tend to forget that procedures and functions have specifications and bodies just like packages and tasks do. Ada requires you to compile the specification of a package or a task before you compile its body. She lets you omit that step, however, when compiling non-generic procedures and functions. We almost always take a short cut when compiling subprograms by compiling simply the subprogram body without compiling the specification first.

Sometimes taking a short cut turns out to be longer, and porting Get_Command_Line to the Meridian environment is an example of such a situation. But I learned my lesson before porting Show to MPX-32. I compiled the package specification shown in Listing 59. There's not much too it, but it saved a mountain of work. After compiling Listing 59, I compiled the Show procedure and my first attempt at the

Get_Command_Line body. It didn't work of course, but when I changed and recompiled it I was delighted that I didn't have to recompile Show. It took me several iterations before I got Get_Command_Line right, but I only had to compile Show once. The correct Get_Command_Line body for APLEX Ada running on MPX-32 is shown in Figure 38.

4.1.6.2. Porting Show to VAX/VMS.

Porting Get_Command_Line to the VAX/VMS environment was the most difficult. First there was the problem of finding a system service that would get the command line. That wasn't a trivial task. DOS is described in three paperback books with a total thickness of about four inches, but VMS is described in a series of big, orange, three-ring binders that take 5 or 6 feet of shelf space. That means there's a lot more haystack to find the needle in. To make matters worse, the needle was cleverly disguised. It was called Get_Foreign, which doesn't immediately suggest suitability for fetching the command line.

Once you find this service, you have to figure out how to interface with it. DEC has some special pragmas, Interface and Import, that allowed me to associate the LIB$GET_FOREIGN service in the system library with the Get_Foreign procedure specification. Listing 60 shows how this was done.

The final problem is using it. If you compile and link it with the Show procedure, it produces an executable module. The normal way to run an executable module is to type RUN SHOW. In this case, however, we want to type RUN SHOW SOMEFILE.EXT. If you do that, it complains about TOO MANY PARAMETERS. I suppose RUN calls LIB$GET_FOREIGN to find out what to run, and is expecting only one parameter. When if finds two, it generates an error message.

The magic VMS trick is to use an alias. If you type $SHOW :== $MY_DISK:[MY_DIRECTORY]SHOW.EXE (where MY_DISK and MY_DIRECTORY represent the actual path to the executable file), then you can type SHOW SOMEFILE.EXT and it will work (because you don't have to use RUN to run the program). It is convenient to put this command in your login file, so you don't have to remember to type it before you try to SHOW something.

This is an awfully brief explanation, but remember it doesn't have anything to do with Ada. These are features of VAX/VMS that are mentioned here just because they were necessary to port the Get_Command_Line procedure to VMS. If you want to know why these things work, take a course on the VMS operating system, or talk to your local VMS wizard. (I'm lucky to have Dave Dent around to find

Figure 38. Get_Command_Line body for Gould Aplex Ada.

```
------------------------------------------------------------
--            GCLBG.ada
--            9 June 1987
--            Do-While Jones

--  Command line interface for Gould APLEX Ada running
--  under MPX-32.

--  This procedure is NOT PORTABLE because it uses
--  a utility package supplied by Gould with their
--  APLEX Ada compiler.

with HOST_LCD_IF;
-- HOST Lowest Common Denominator InterFace.
procedure Get_Command_Line(TAIL   : out string;
                           LENGTH : out natural) is
  TEXT : string(1..80);
  N    : natural;
begin
  HOST_LCD_IF.Get_Param_String(TEXT,N);
  LENGTH := N-1;
  S(1..N-1) := TEXT(1..N-1);
end Get_Command_Line;
```

these VMS features for me.)

4.1.7. Library Procedure Summary

You can separately compile a single library procedure or function without having to put it in a package. (Most people must not know this because several times I've seen package specifications with nothing but a single subprogram specification in it.) When you do this, you freeze the interface. Then you can recompile the body over and over again, and Ada will check to make sure you have used exactly the same formal parameter list. As long as you don't change anything in the parameter list, you can make as many changes in the body as you like without making units that depend on the specification obsolete.

4.1.8. Common Command Names

Remember how I thought typing MORE SOMEFILE.LST caused DOS to crash? We are about to run into that same problem again, and this time we will see that command names can sometimes get you into trouble.

I was particularly frustrated because I was using so many different operating systems. They all used different names for deleting files. I could never remember if I should ERA, DEL, DELETE, rm, KILL, or VOLMGR. On one systems LIST would type a file, on another it would display the directory. It was driving me nuts! I decided I wanted to try to standardize utility names. Since I was using UNIX then, I decided to rename the Show program to More to match the UNIX name. Since it was going to have a UNIX name, I also wanted it to work like the UNIX version.

4.2. The More Tool

The UNIX more command pauses after each screenful (22 lines), printing "--More--" at the bottom of the screen. If the user types a carriage return, one more line is displayed. If the user types a space, another screenful is displayed. If the user types an integer, that number of lines is printed. If the user hits d or CONTROL-D, 11 more lines are displayed. The UNIX version also has nine command line switches, but since I've never used any of them I didn't bother to implement them. (That's left as the proverbial exercise for the reader. See the UNIX documentation for a description of the nine unimplemented options.) The More program is shown in Listing 61.

When I tried to run More on the PC, I ran into a name clash. DOS already has a command called MORE. It differs from the UNIX MORE because DOS MORE has to be used as a filter. That is, you can say TYPE ANYFILE.EXT | MORE and it will show you a screen full at a time, but if you type MORE ANYFILE.EXT it just sits there waiting for you to enter data from the keyboard, which it will display one screen at a time. When that happened I thought the system crashed.

I could have left the name of my program Show, but that doesn't solve the problem. I had fallen into the habit of typing more ANYFILE.EXT on the UNIX system, and occasionally did that on DOS. So, I used the DOS REN command to rename MORE.EXE to DOSMORE.EXE. Then, when I ran my version of More it did not clash with the existing DOS program of the same name. If I want to use the original DOS MORE program, I can still enter a command such as TYPE ANYFILE.EXT | DOSMORE.

The moral of the story is that users expect certain results when they do certain things. It is hard for them to remember that the same command does different things depending on the context. Sometimes users can work around the problem by renaming commands or creating aliases, but they shouldn't have to. Whenever you write a program that has a direct interface to a human user, be sure you are consistent with what that user is accustomed to.

4.2.1. Multiple Loop Exits

Respectable loops have one entry and one exit point. To be perfectly proper the exit point is at the beginning or end of the loop. Some people allow the exit point to be in the middle of the loop, but they do so at the risk of being snubbed by elite programmers. A close inspection of Listing 61 shows that it ends with a loop containing (shudder!) two exit points, and one of them is in the middle. How could anyone with a name like Do-While do such a thing?

The software guideline restricting all loops to Do- While or Repeat-Until structures has merit. Certainly I'm not advocating a return to long loops full of GOTOs that twist a tangled trail through tortuously tightened tentacles. Those structures are as hard to understand as they are to read. Loops that enter at the top and exit only at the top or the bottom are much easier to understand and maintain than something that looks like a nervous giant squid.

I am prepared to argue, however, that there are special instances when the most maintainable loop has multiple exits. Furthermore, I submit that the More program is one such instance. I think it is much cleaner than the single exit from the nested named loops in the Show program.

The two-exit loop in the More program more closely describes what is really happening than the nested loops in the Show program do. Specifically, it displays some lines of text on the screen, and if it has displayed all there are to display, it quits. If there are more lines left, it lets the user decide how many more he wants to see. If he doesn't want to see any more, the loop ends. The loop is so short, it is easy to find all the exit points.

If I had coded the Display and get routines in-line instead of using procedure calls, then I would agree that the two exits are hard to find, and would submit to any number of lashes with a wet noodle. Instead, the Display routine hides the confusing fact that there is another loop reading and writing one line at a time. The Show procedure might have less trouble passing a code walkthrough, but everyone should agree, More is better.

4.3. The Write Tool

It seems that every time I get a new, better, computer, it is always harder to do the things I used to do so easily on my old, obsolete computer. On my old CP/M computer I had an 8080 assembly language program called Write that printed files on the printer. It put a title at the top of the page, a page number at the bottom, and never printed on the page perforations. It would double space the listing, too, if I asked it to. I was disappointed when I discovered I couldn't do the same thing on my "advanced technology" clone.

The DOS PRINT command prints files, but it just copies them straight to the printer, with no headers or page numbers, and doesn't even skip over page perforations. I bought a well-known word processor from a major software house, and thought I could use it to print flat ASCII files. I can, but it sure is a nuisance. The command sequence to print a single-spaced file is:

```
<ESC> TRANSFER LOAD FILE.EXT <CR>
<ESC> FORMAT DIVISION PAGE-NUMBERS YES <CR>
<ESC> FORMAT DIVISION MARGINS 0 <TAB> 0
<TAB> 0 <TAB> 0 <CR>
<ESC> PRINT PRINTER
<ESC> QUIT NO
```

That doesn't put titles on the top of every page. (I know there must be a way to get titles, but I haven't figured out how yet.) If I want to double space it I have to do all of the above, plus

```
     <SHIFT> F10
     <ESC> FORMAT PARAGRAPH <TAB> <TAB> <TAB> <TAB> 2 <CR>
```

somewhere in the middle.

Oh! how I longed for the good old days when I could WRITE FILE.EXT. It didn't take too long for the frustration level to build to the point where I translated my Write program from 8080 assembly language to Ada. Figure 39 shows the first version of the Write program, and Listing 62 shows the final version.

4.3.1. Error Recovery and Help

Figure 39 was a direct translation from assembly to Ada. It didn't have any help features because I wrote the program for my own use and didn't need them. When I decided to publish it, I knew other users might press the question mark key to get help, so I had to include NEEDS_HELP handlers. I suppose I could have put one big help procedure at the end of the program that explained every option, but then the user would have to read all the answers and figure out which one answers his question. I prefer to give the user short, pertinent explanations that help solve the immediate crisis.

What managers and programmers often fail to realize, is that a major portion of a program with a user interface will be devoted to error recovery and help messages. Everyone expects a routine that asks the user for the first page number to include some statements that convert a string to an integer. They often don't realize there must also be some statements that know what to do when the user says the first page number is "banana". Look at how much of the Write program is devoted to error recovery! Compare Figure 39 to Listing 62 to see how short the program could be if I left all the error recovery routines out.

The final typesetting process may change the exact number of lines in these two versions of the Write program, but before reformatting them to make them fit on the printed page, the original version was 57 lines and final version was 145 lines. That means 57 of the lines in Write are doing the work, and 88 lines are just there for error recovery. That's a 154% increase in program size to make the program user friendly. The 154% figure isn't an absolute constant. The amount of expansion depends on the number of questions you ask the user, and how verbose each help message is, but it isn't unusual for programs with extensive user interfaces to more than double in size when error recovery and help messages are added. Take that into consideration when you are estimating the size of a software project, and don't be naive enough to think that error recovery will be 2% of the total software effort!

4.4. The Line Tool

Meridian now has an Ada Development Interface (ADI) that integrates a smart editor with the compiler. If you compile a file that has error in it, you can push a button and the cursor will move to the point in the source code where the next error is. AdaVantage Version 1.5 didn't have that feature. If you compiled a file that contained errors it would write a message to the screen telling you that line number X in SOMEFILE.ADA contained a certain error. You could call up the EDLIN editor and go to line X to see what it was. Sometimes the error was really in line X-1 (a missing semicolon, for example), and the compiler didn't recognize the error until line X. Sometimes the error was many lines earlier. (For example, proper use of a variable that was improperly declared generates an error message pointing to the line using the variable, not the line declaring it.)

```
Figure 39. Write without help and error recovery.
-----------------------------------------------------
with Get_Command_Line;
with SCROLL_TERMINAL, SCROLL_PRINTER, TEXT_IO;
procedure No_Help_Write is
  TITLE               : string(1..79);
  FILENAME            : string(1..68);
  LENGTH              : natural;
  FILE                : TEXT_IO.File_type;
  PAGE                : positive;
  DOUBLE_SPACED       : boolean;
  LINES_LEFT_TO_PRINT : natural;
  TEXT                : string(1..250);
  RESPONSE            : character;
  PAGE_NO             : string(1..4);

begin
  Get_Command_Line(FILENAME,LENGTH);
  TEXT_IO.Open(FILE, TEXT_IO.IN_FILE, FILENAME(1..LENGTH));
  TITLE := (others => ' ');
  TITLE(1..LENGTH) := FILENAME(1..LENGTH);
  SCROLL_TERMINAL.put_line("Enter page TITLE, please.");
  SCROLL_TERMINAL.get("", TITLE, TITLE);
  SCROLL_TERMINAL.get("Start numbering pages at page ",
   "1",PAGE_NO);
  PAGE := integer'VALUE(PAGE_NO);
  SCROLL_TERMINAL.get("SINGLE or DOUBLE spaced? (S/D) ",
   'S', RESPONSE);
  case RESPONSE is
    when 'D' | 'd' =>
      DOUBLE_SPACED := TRUE;
    when others =>
      DOUBLE_SPACED := FALSE;
  end case;
  loop
    exit when TEXT_IO.End_Of_File(FILE);
    SCROLL_PRINTER.new_line(4);
    SCROLL_PRINTER.put_line(TITLE);
    SCROLL_PRINTER.new_line(3);
    LINES_LEFT_TO_PRINT := 50;
    loop
      exit when TEXT_IO.End_Of_File(FILE);
      TEXT_IO.get_line(FILE,TEXT,LENGTH);
      SCROLL_PRINTER.put_line(TEXT(1..LENGTH));
      LINES_LEFT_TO_PRINT := LINES_LEFT_TO_PRINT-1;
      if DOUBLE_SPACED then
        SCROLL_PRINTER.new_line;
        LINES_LEFT_TO_PRINT := LINES_LEFT_TO_PRINT-1;
      end if;
      exit when LINES_LEFT_TO_PRINT < 1;
    end loop;
    SCROLL_PRINTER.new_line(LINES_LEFT_TO_PRINT+3);
    SCROLL_PRINTER.Set_Col(30);
    SCROLL_PRINTER.put_line(integer'IMAGE(PAGE));
    PAGE := PAGE+1;
    SCROLL_PRINTER.new_page;
  end loop;
  TEXT_IO.Close(FILE);
  SCROLL_TERMINAL.put_line("Done.");
end No_Help_Write;
```

I decided I would like to be able to type LINE X SOMEFILE.ADA and see that line on the screen. The more I thought about it, the more I realized I really wanted to see a few lines before and after line X. So, I wrote the Line program shown in Listings 63 and 64.

Although the Line program was written in response to a specific need (to display lines containing errors), it isn't limited to that single use. You can use it to browse through any file for any reason.

4.4.1. Multiple Arguments

The Line program needs two arguments on the command line (a line number and a file name). The Extract subunit extracts these two pieces of information from the command line and returns them as separate parameters. This forced me to make some design decisions. Which argument should come first? What should I do if the user enters the arguments in the wrong order?

My first reaction was that the file name should be the first argument and the line number should be the second one. This makes sense from a programmer's point of view because the file has to be opened before the program can start looking for a particular line. It also seemed consistent because More and Write both have filenames immediately following the command.

I wrote the Extract routine to take the file name first, used the program for a while, and found I kept making mistakes. If I wanted to see line 15 of SOMEFILE.ADA, I naturally typed LINE 15 SOMEFILE.ADA. It seemed wrong to type LINE SOMEFILE.ADA 15. (Since it's usually right to not split an infinitive, analogy suggests that I shouldn't separate 15 from LINE.) From a user's point of view, it makes more sense from a user's point of view to put the line number first, so I changed the order in Extract.

4.4.2. Error Tolerance

I put the arguments in the order that seems right to me. Who's to say that every user will feel the same way. It may be natural for them to put the file first. They may naturally enter the arguments in the wrong order. That's an easy error to detect. Only one of the fields will be the image of an integer, so that field must contain the line number. If the program detects the incorrect entry, the program could respond with this error message, "You have entered the arguments in the wrong order. The correct syntax is LINE N FILENAME.EXT."

The user might respond to this error message with a respectful genuflection, and say, "Oh, I'm so sorry. Please forgive me. I promise never to make that mistake again." Users like that may exist, but I've never found one. Most users will say, "If you're so darn smart, and know that the arguments are reversed, why don't you just do what I want?" The point is well taken. If a program is smart enough to recognize the error, and can tell the user exactly how to correct it, can't it just as well be smart enough to fix the error itself?

This kind of tolerance is almost unknown in user interfaces, but perhaps the Line program may start a trend. Instead of insulting the user, it simply complies with any poorly stated command. If the user enters the two arguments in the correct order, it works. If the user enters the two arguments backwards, it works. If the user forgets to enter the file name, it asks for the file name. If the user forgets to enter the line number, it asks for the line number. If the user doesn't enter either, it asks for both. There's no need to be nasty when its so easy to be nice.

4.4.3. Presuming Too Much

If a software component is going to be reusable, it can't presume too much about how it will be used. A case in point is the Get_Command_Line procedure for the Meridian system. It presumes too much and that makes it awkward to use.

The Alsys and DEC versions of Get_Command_Line are built on general routines that return the whole command line. Alsys and DEC didn't make any assumptions about how anyone would want to use the command line. They just give you the command line and let you do whatever you want with it.

Meridian assumed that anyone who wants the command line will want to break it down into individual arguments, so they tried to do the programmer a favor and split it apart for him. Their ARG package tells you how many arguments there are, and passes an array of arguments back. That's really handy if that's exactly what you want to do. If it isn't then it is clumsy.

I've written a routine that doesn't appear in this book (because it doesn't teach any new lessons) called Search that searches a file for a text string. It is invoked by the command SEARCH FILENAME.EXT "A STRING WHICH MAY CONTAIN PUNCTUATION, SPACES, AND WHO KNOWS WHAT ELSE!". The Meridian ARG package returns this as 13 individual arguments, but there are really only two (FILENAME.EXT and thetext string). I have to go to the trouble to put together the things that Meridian has taken apart.

That's not a terrible ordeal. Listing 65 shows how to do it. It's inefficient for ARG to take apart the command string and then make Get_Command_Line put it back together so Line.Extract can take it apart again, but that's the price you pay for portability if you make your utility routines too specific.

4.5. The Search Tool

Let's talk about the Search program some more. The command line contains a file name and a text string. The Search program searches that file for every occurrence of the string, and prints a list of all the line numbers containing that string. It's a handy tool for authors who are building the index for a book. Quality assurance folks can use it to search through code for gotos and abort statements. Students can use it to search through source code to find examples of particular Ada constructs. After Search gives you a list of line numbers, you can use Line to look at those lines in context.

I can see I have created an insatiable desire to have this marvelous tool. You can't wait to turn to the back of the book to find the source listing. Don't bother. It isn't there.

4.5.1. I've Seen That Before

I took the Search program out of the text because I thought most readers would say, "I've seen that before!" You must have noticed the striking similarity of More, Write, and Line. They are really three variations of one basic program (Show). They all take data from a command line, open a file, and display the contents in a slightly different way. Search is just a fourth variation on the same theme. The only difference is that it contains a boolean function that tells if a string is contained in a line. If you can write that function, then you can edit Line into Search without much difficulty. Why don't you try it and see for yourself?

4.5.2. Keep Selling the Same Software

It isn't unusual to find yourself in a job where you keep doing essentially the same thing over and over again. If you ever write one missile simulation it is a good bet you will write more in the next year or two. If you write a compiler program, there is a good chance your boss will tell you to write another one (for another language or the same language on a different computer) as soon as you get finished. Rarely does one work on a compiler one day and a missile simulation the next. Companies and individuals keep selling the same basic product.

When I wrote the More, Write, and Line programs, I was able to do them quickly because I already developed all the building blocks I needed by the time I finished the Show program. This is what modularity and reusability is all about. I almost always start a program by making a copy of an existing program, deleting parts I don't need, and adding some new code. If you find yourself starting from scratch every time you write a new program, you are doing something wrong. Whenever you start a new program you should ask yourself, "What program have I previously written that I can use for a starting point?" You ought to be able to think of several possibilities, unless you are doing something totally new. (You can't turn a compiler design into a missile simulation, but you should be able to edit a compiler into pretty printer much faster than you could write a pretty printer from scratch.) If the answer to your question is, "Well, I could use program X, but it would take much too long to modify it.", then you deserve to be beaten severely for the bad job you did on program X. It shows that program X is not modular or maintainable.

Of course the ultimate in programming excellence is seen when everyone in your group writes such good code that you can use each other's code. Some people say that will never happen, but I don't think it is unrealistic to expect people to be able to write code good enough to share with a friend. I get by with a little help from my friends. (Dent, Leif, and Lucas-- in alphabetical order.)

The way to make big money and impress your boss with your tremendous productivity is to build on what has already been done. People will say, "Isn't that amazing! Jones wrote the More program, and the next day he wrote the Write program, and the day after that he wrote the Line program, and on the fourth day he wrote the Search program. Four different programs in four days! What a genius!" The truth is I wrote one program and sold it four times. As long as I keep my mouth shut, everybody is happy and I'm rich.

4.6. Draw_Poker, Version 2

The Draw_Poker program developed in this case study is an improved version of one I published several years ago. This program was developed using a more rigorous design strategy than Show, More, Write, and Line used. It was developed using Software Engineering.

4.6.1. Software Engineering

Software engineering differs from programming in several areas. Software engineering requires more planning than ad hoc programming. A wise person once said, "If you don't know where you are going, you will wind up someplace else." This is especially true of software engineering. You really have to know where you want to go.

Of course, just knowing where you want to go isn't enough. You also have to know how to get there. Software engineers use a methodology to arrive at the goal.

> Meth-od-ol-o-gy. 1 : a collection of methods, rules, and superstitions used as a substitute for intelligence. 2 : a particular procedure or set of procedures used to turn a difficult problem into an impossible one.

Software engineers are often devoted to their favorite methodology with a zeal that exceeds the enthusiasm of all but the most fanatic member of a religious order. That makes it difficult to carry on a rational discussion about specific methodologies.

The primary difference between software engineering and programming is the level of documentation. Software engineering requires you to document (1) requirements, (2) analysis, (3) design decisions, (4) error reports, (5) test results, and (5) configurations.

Some of these topics fall outside the scope of a book about Ada, but we can at least touch on some of them, particularly when we talk about how they affect Ada.

4.6.2. Military Standards

The current military standard for Defense System Software Development is MIL-STD-2167A. It supersedes MIL-STD-1679. I'm not going to say much about those two specifications, because this book describes techniques I have found to be valuable. Those specifications have good intentions and present a theoretical method for software development, but in practice, I've never seen them do anything but increase cost, cause schedule delays, and result in inferior software.

There will certainly be some who will say that the development of the Draw_Poker program does not satisfy the requirements in 2167A. My response to that charge is, "That's right. I make no attempt to satisfy those requirements."

4.6.3. Goals and Requirements

The first thing you have to do is recognize the difference between a goal and a requirement. The goal is what you want or need to do. A requirement is an intermediate objective that must be achieved to reach the goal. I am amazed at how often people focus on requirements and lose sight of their goals.

I first realized how dangerous it is to confuse requirements with goals one time when I was reviewing a proposal for a target detecting device. The engineer presenting the proposal began his speech by saying, "The requirement is to design a complicated, multi-level interrupt-driven microprocessor-based target detecting device." I interrupted him, saying that I couldn't believe that was his requirement, but he insisted it was. I asked him if his sponsor in Washington said, "I don't care what kind of targets this thing detects, as long as it is complicated and uses multiple levels of interrupts!" He assured me that's what the sponsor demanded. I couldn't convince him that the goal was to detect certain kinds of targets in a certain environment.

It may be that the problem can be solved using an interrupt-driven microprocessor, and there might several levels of interrupts, and it might be complicated, but that shouldn't be a requirement. Suppose someone thinks of a simpler, cheaper, more accurate, more reliable solution that only uses a single-level interrupt. Do you want to exclude that solution because it doesn't meet the requirements? I don't!

The goal is absolute. It represents a need that must be satisfied. Requirements reflect the current thinking of what intermediate steps must be taken on the road to achieving the goal. Requirements can (and should) change if a better way to achieve the goal is discovered.

I believe the first step in software engineering is to document the goal. To the mainstream, respected software professionals, this means writing a software specification. Although I agree with this in principle, I've found that writing a software specification doesn't help much. Software specification tend to focus on requirements, not goals. This often locks you into a requirement that is counter- productive to the goal. Furthermore, the software specification tends to be written in legalese. In theory it is so precise it can be interpreted only one way by a court of law. In practice it is usually so complicated that nobody really understands it, and everybody thinks it means something different (especially the customer and programmers). Typically the customer isn't happy with the final product, and the blame eventually falls on the party with the poorest lawyer.

I set design goals by writing a sales brochure and a user's manual. The sales brochure emphasizes the goals (what it does, why you need it, and what it costs). The user's manual explains how it works. Most people wait until the project is over before writing these documents. Then they realize too late that the product doesn't do what the customer needs, is more expensive than existing products, or is too complicated to use.

Whenever I pick up the manual for my word processor, I ask myself, "Did someone write this user's manual before the product was designed? Did they maliciously intend it to be this difficult to use?" I suspect that some managers made a list of requirements consisting of every feature they could think of. Then a hoard of programmers did whatever they had to do so they could claim they met all the requirements. Then some unfortunate soul got stuck with the job of trying to write a manual to explain how to use it. (Of course the technical writer gets all the blame for the lousy manual.) Consider this: If you can't easily explain how to use your program, how can any user be expected to learn to use it? The user's manual sets the requirements for the user interface. You have to write the user's manual first, or who knows what mess you will end up with.

4.6.4. Case Study

Using an example, let's show how the sales brochure and user's manual relate to Ada program development. This case study assumes that we want to get into the business of selling video card games. Our first product is going to be a video draw-poker gambling machine. (Later we could expand the product line to include blackjack, bridge, canasta, and maybe even old maid.) The first step is to imagine the product is done. We need a sales brochure describing it and a user's manual that tells how to play it.

4.6.5. Sales Brochure

What does the sales brochure have to say, for us to be able to sell our product? It should rave about the attractive graphics that encourage players to gamble because it is fascinating to watch the cards being shuffled and dealt. It would be a big selling point if we could say that it accepted coins or paper money of any denomination, so players won't stop gambling just because they run out of quarters. That would also let players bet variable amounts, which encourages them to try a "system" they think will help them beat the house. The minimum bet the machine will accept is $1, so penny players won't keep the machine busy while real gamblers are waiting to use the machine. The maximum bet is $999 for safety. (You have to be suspicious of anyone willing to be $1,000 or more on a coin operated machine.) The machine should shuffle the cards before every hand to prevent players from gaining an advantage by counting cards.

Perhaps the most important section of the sales brochure is the cost analysis. It tells prospective customers the average amount of money gambled per hour, the house advantage, and the resulting income per hour. When this income is compared this with the cost of the machine, it shows how quickly it pays for itself. Then a comparison of the expected income to the anticipated monthly maintenance shows how much money the machine makes per day. Every day the customer delays in purchasing the machine, he loses that much money. He can't afford not to buy it!

The cost analysis is just as important to us as it is to the customer. It tells us the maximum amount we can sell the machine for. If our sales brochure has to say that it takes 10 years for the machine to pay for itself, no rational person will buy it. We will have limited the market to people who are too stupid to realize it is a bad investment. Knowing how much we can get for each machine, the number of machines we can produce per month, and the amount of profit we need to make each month, we can figure how much profit we must make on each machine. When we subtract the profit from the selling price, that gives us the production cost. If we can't produce the machine at that price (including nonrecurring engineering costs), then there isn't any point in even starting the project.

The sales brochure should also include hardware features (vandal resistant, won't accept slugs, silent alarm when theft attempts are detected, and so on) that we will ignore here since this is a book on software. You can't ignore those aspects in the real world.

4.6.6. User's Manual

The user's manual tells how the machine will work. We have to remember that someone who has had too much to drink and has gotten discouraged playing the roulette wheel, isn't going to pick up a one hundred page manual and read it before playing a draw poker game. The user's manual has to be printed in a few large words on the face of the machine. It has to be short and sweet. Something like this:

```
Put in as much money as you want to BET.
Press DEAL.
Press the button under each card you want to DISCARD.
Press DEAL again.
If you win, THE MACHINE PAYS YOU!
Play again, It's fun!
```

You also need pictures showing winning hands and how much they pay off.

4.6.7. Checking the Requirements

Periodically during the course of the design you should check your design against the sales brochure and user's manual. Are they still accurate descriptions of your product? If not, try changing the documents to match the design and see if you still have a viable product. If so, continue with the product development. If not, don't continue on this path because you will just wind up spending more money to design a product you can't sell. Change the design to match the original descriptions. If you find it isn't possible to design what you originally described, or something close enough to your original idea that is marketable, then quit now before you waste any more money.

I think this is a major cause of defense contract overruns. Checking a design against a software specification doesn't tell you much. It just tells if requirements are being satisfied. That doesn't tell you if you are meeting your goals or not. If you aren't meeting the requirements, the tendency is to lower them so you can meet them. That doesn't help. You just spend more money to build something that's inadequate.

4.6.8. Planning for Reuse

Of course, it isn't enough to just set goals and periodically check to make sure you are still heading toward them. You have to make progress at a fast enough rate that you achieve the goal in your lifetime. An important part of software engineering is learning how to cut a big project down to size, so it is possible to complete it on time and under budget. One way to do that is to reuse code you've previously written and tested before. If you are lucky, you might find something lying around that you can use, but that seldom happens. The best way to reuse software is to plan ahead and create components you are likely to be able to use in the future.

Let's return to the Draw_Poker example. If this is to be the first of a whole line of video card games, then it seems likely that some of the routines we develop for this game will be useful on other games. For example, routines that shuffle, deal, and display playing cards aren't limited to poker. These routines are in the PLAYING_CARDS package (Listing 66), so they can be reused in other card gmes.

If we were writing this project in FORTRAN or assembly language, we might recognize the utility of general-purpose routines that shuffle, deal, and otherwise manipulate playing cards. We would be wise to collect them in a file and compile (or assemble) them into object code modules that could be saved in a library. You might think that the PLAYING_CARDS package is just like one of these files of library routines. Well, in some ways it is, but it is really much more. What makes the PLAYING_CARDS package different from a FORTRAN or assembly language library is the fact that those libraries contain nothing more than executable routines. If you look in the PLAYING_CARDS package specification you will find routines, but you will also find abstract data types, constants, and error conditions. This makes it more complete (and therefore more useful) software component than a simple library of general-purpose routines.

4.6.9. Abstract Data Types

If I were going to write card playing programs in FORTRAN or assembly language, the first thing I would have to do is decide how to represent individual playing cards. In those languages I would be limited to a few standard data types. The two logical choices are strings and integers. (I think I can safely rule out real numbers without too much consideration.) If I chose to use strings, I might represent the four of diamonds as "4D". If I chose to use integers, then I might use the numbers 1-52 to indicate the 52 cards in the deck, or I might let the numbers 102-114 represent two through ace of clubs, 202-214 represent two through ace of diamonds, and so on. How I choose to represent cards will greatly affect how easy it is to sort cards; shuffle them; check for a flush; check for two-, three-, or four-of-a- kind; or tell which card wins a trick. A poor choice of data representation will make solving the problem much more difficult. Human thought will be diverted from the main problem (how to play poker) and be wasted on an artificially created problem (how to force an integer or string to have the properties of a playing card).

Furthermore, the decisions I make about data type representations will greatly affect how I write my executable routines. If I change the representation of playing cards late in the program, then I'll probably have to throw away everything I've done already.

Ada's abstract data types take the burden from you and put it on the compiler. She lets you make the data types fit the problem, rather than trying to change the problem to fit the available data types. Then she let's you develop algorithms that don't depend on how the data is represented. This leaves you free to change the representation at any time without losing much (if any) of the work you've already done.

Your initial reaction to abstract data types might be influenced by your emotional reaction to the term "abstract." If your immediate reaction to the term "abstract art" is, "A confusing picture that's distorted and hard to understand," then you probably are a little afraid of abstract data types. You will expect them to be weird and hard to understand. Well, don't let a few bad artists scare you off.

A good abstract artist has the ability to separate the important features from meaningless ones. Certainly an abstract artist distorts reality, but a good one does so in a way that emphasizes the important points and makes the trivial points disappear. If this is properly done, it isn't confusing at all. In fact, it conveys meaning to the spectators with remarkable clarity because the message shown by a few powerful images without the clutter of extraneous details.

The same thing is true of the abstract of a technical paper. The abstract describes the paper by concentrating the important facts in a small space, without cluttering the description with a lot of minute details.

An abstract data type does the same thing an abstract painting or the abstract of a paper does. It represents the important characteristics of the object without cluttering it up with the unimportant details (details like how many bits are used and how the bits are encoded).

If an abstract artist wanted to capture the essential details of a poker game on canvas, what would he do? He would watch the game, and mentally decide what was important to the fundamental activity. He would mentally eliminate the table from the picture because it isn't necessary to the game. Sure, it keeps the cards from falling on the floor, but it doesn't affect who wins, so it isn't an important part of the picture. The only important objects the artist would include in the picture are the cards that were dealt, the players' hands, and the deck. It doesn't matter what shape, size, or color the cards are, as long as you can tell what rank and suit they have. Ranks and suits are important abstract qualities of cards. The thickness of the card and design on the back aren't important. The artist is free to represent a card in any manner, just so long as you can tell what suit and rank it has.

The actions that are important are the shuffling of the cards, dealing of the cards, discarding the cards, and determining the value of a hand to see who the winner is. The skillful artist must figure out how to show these dynamic actions happening on a static piece of canvas.

The artist may or may not show the cards being sorted. Sorting the cards in a hand is an optional action. It makes it easier to see if a hand holds a winning combination, but you could figure that out without sorting the hand if you had to. Who knows, maybe there is a way to hash code the values in a hand, that allows you to tell a winning hand from a losing hand quicker than you could if you sorted it. You are probably wise to sort a hand, but it isn't a requirement.

The PLAYING_CARDS package specification (Listing 66) is just an abstract painting in words instead of oils. It describes important objects and actions, and eliminates all the other details.

Look near the beginning of the package specification. I've told Ada to create a data type called Suits. Objects that are Suits can have values of CLUBS, DIAMONDS, HEARTS, or SPADES. I don't care how Ada represents these things internally. She can use 0, 1, 2, 3 or 1, 2, 3, 4, or 'C', 'D', 'H', 'S', or anything else she desires. It doesn't matter to me, as long as she is consistent. I have, however, given her an implied precedence. CLUBS is the lowest value and SPADES is the highest. Similarly I've defined an ordered set of values called Ranks.

Then I have defined three data types that are even more abstract. Cards, Hands, and Decks are private data types. We don't know what values they can have or how they are represented, but we really don't care about those details at this level of the program.

Skipping over the three exceptions for the moment, we see the things we can do with these abstract objects. We can find out the suit and rank of a card. We can open a new deck or shuffle it. We can open a new hand, sort it, peek at each individual card, play any card, tell if a card has been played, see if a hand is full, or deal a card to a particular hand.

Returning to the exceptions, I have defined three things that could go wrong. When you take the cellophane off a box of cards and inspect it, you might find an extra seven of hearts, or discover that the two of spades is missing. This shouldn't happen, but you should know about it if it does. You don't want to force every application program to check for it, so the Open_New action does it automatically. A more likely error is that an application program will get carried away and try to deal a 53rd card from the deck. If the application program is careless enough to do that, it probably isn't doing any special error checking for that condition. The Deal action better take responsibility for checking for that error. By the same reasoning we can conclude that an application program may try to deal a card to a hand that is already full, and not check for that error. That's why I included these exceptions in the package. They remind me that these are error conditions I must check for when I write the package body, and tells whoever uses this package what error flags might be raised.

These are all the things we need to do to all the objects we need for a poker game. To be truly reusable we should also include objects needed for other card games (tricks and trumps for bridge, melds for canasta, and so on), but I didn't want to complicate the package with unnecessary objects or operations.

4.6.10. Private Types

I chose to use private types to represent Cards, Hands, and Decks. I could have used visible records and arrays or limited private types. I chose not to. There were good reasons to use private types, and those reasons are worth an explanation.

Suppose I had used visible records and arrays. (That is, suppose I had moved the type definitions from the private part to the place where the three "... is private;" declarations are. That would have eliminated the need for the Suit_Of and Rank_Of functions because application programs could simply use CARD.SUIT and CARD.RANK. That is exactly what I wanted to avoid. By making the definition of Cards private, I can be sure that I can change the definition from a record to a simple integer if I like, and it won't affect any other part of the program.

You say, "Why would you want to change the representation of Cards to an integer?" Well, suppose I discovered that my program wouldn't fit in memory. Looking at the code generated by the compiler I see that a card is represented by a two component record. The first component, the SUIT, is a number 0 to 3 represented as a 32 bit integer. The second component, the RANK, is a number 0 to 12, also represented by a 32 bit integer. Therefore it takes 64 bits (8 bytes) to represent a card. Since there are 52 cards in a deck, the representation of those 52 cards takes 416 bytes. If I represented a card as a number from 0 through 51, then I could use 1 byte per card, and only 52 bytes per deck. That's a storage savings of 88%!

If I change to an integer representation, and I have used private types, all I have to do is rewrite the Suit_Of and Rank_Of functions. That's easy enough to do. I can use integer division by 13 and the VAL attribute to get the SUIT, and modulo 13 operator and VAL attribute to get the RANK. I can recompile any program that uses PLAYING_CARDS and I can be sure it will still work. (It will take less space, and may run slightly slower, but it will still work.)

If I change the representation of a card from a record to a 1-byte integer, and have used visible types without the Suit_Of and Rank_Of functions, then I will have to find every line of every application program that contains CARD.SUIT or CARD.RANK. Granted this is easy to do with a text editor, and Ada will tell me if I missed any, but I still have to insert division and modulo operators all over the place. There is a good chance I will make a mistake doing that.

So visible types aren't a good choice. But if private is good, then limited private must be better. Isn't it? Well, the decision between private and limited private isn't always easy. If you immediately see a reason why you will want to assign values to an object, or need to check to see if two objects are equal, then you can't use a limited private type. (Limit private types don't have assignment operators or equality tests.) If you don't see a need to do these things, it is better to try to use a limited private type.

Rather than spending a lot of time and effort figuring out if I need private or limited private, I usually just try limited private first and see if that leads me into trouble. I couldn't see any reason to assign a value to Cards (I wasn't going to put any cheating in the game), nor did I see a need to check two Cards for equality (there's only one deck, and every card is unique), so I decided to use limited private for Cards at first. It didn't appear that I would need assignment or equality for Hands or Decks either, so I made them limited private, too.

I ran into trouble when I tried to make a copy of a hand and sort the copy. Then I realized that I did have a legitimate need to assign a value to a hand. I made Hands private instead of limited private. Then I realized I might want to make a copy of a deck for a duplicate bridge game so I would need to assign values to decks, too. Then I realized I might not want to wait until I randomly dealt myself a royal flush to see if the royal flush detection algorithm really works, and so I might want to assign particular values to Cards in my hand in a test routine. There went the last limited private type.

I think it is a good idea to use limited private types whenever possible, so I always try them first. If they don't work, it is a simple matter to strike the word limited with a text editor. If I start out with a private type first, I might not realize I don't need to assign a value to it or test it for equality, and I might leave it private when it should be limited private.

4.6.11. Keep I/O Routines Separate

Some of you may have read an article I wrote for the February 1986 issue of Dr. Dobb's Journal of Software Tools. It contained a Version 1 of the PLAYING_CARDS package. That version contained overloaded Put procedures that I have removed from the version 2. That's because I realize now that it was a mistake to mix I/O routines with processing routines.

I used to believe I should always include I/O routines for new data types in the package that defines the data types. I thought, "There's not much point in creating things if you can't input or output them." Well, that's true, but it doesn't mean those I/O routines have to be in the same package. Version 1 used TEXT_IO to write phrases like "Three of Hearts" to the screen. If I ever write a book of advanced Ada examples, I will probably expand on this example by using a graphic interface to draw the cards on the screen. TEXT_IO wouldn't be any use in that application. I shouldn't have to modify and recompile the PLAYING_CARDS package just because I'm using a graphics package instead of TEXT_IO to display Cards. If I change the output device or output method, I expect to have to rewrite I/O packages, but I shouldn't have to rewrite any processing packages. Routines that turn pixels on and off (or write messages to the screen) have nothing to do with routines that shuffle and deal cards. Therefore, they don't belong in the same package.

4.6.12. A Limit to Reuse

The PLAYING_CARDS package body is shown in Listing 67. It's simple enough that it doesn't require much explanation, but I do have to justify my Sort routine. People have spent years searching for the ultimate sort routine, and here I've used this simple bubble sort. Isn't this a golden opportunity to reuse a generic sort routine?

I'm all for reuse (the Draw_Poker program reuses RANDOM_NUMBERS, STANDARD_INTEGERS, MONEY, DIM_INT_32, SCROLL_TERMINAL, and ASCII_UTILITIES), but sometimes reuse is more trouble than its worth. It's true, a more exotic sort routine might be faster, but how long can it take to sort five cards, even using the most inefficient routine? If I were sorting 5,000 cards, and speed were important, then I might instantiate somebody else's super-optimized generic sort package. In this case it was quicker to write thirteen lines of simple code than search for a reusable component that will do the job.

If I already had a generic Sort routine that was easy to instantiate, and had established its reliability, of course I would have used it. If I expected to need to sort large collections often, then it would make sense to write (or buy) a generic sort routine, verify it, and use it in whenever I needed it. But this is the first time in 22 years of programming that I've ever needed a sort routine, and I don't anticipate needing one again in the next 22. A reusable sort routine isn't high on my priority list right now.

Writing a book is a lot like doing a real project. There is a deadline that has to be met, and you can't waste your time searching for the most elegant solution when you already have something that works perfectly well, especially when there are other things that aren't done yet. Looking for a generic Sort routine is counterproductive.

4.6.13. Efficiency VCR Verifiability

The Sort routine also brings up another issue. Which is more important, efficiency or verifiability? The answer depends on the situation. The Draw_Poker program runs so quickly I had to add some delay statements to slow it down (I like to keep the player in suspense while the cards are being dealt), and it fits easily in memory so I'm not concerned about size. In this situation I don't care at all about efficiency, but I want to be sure the program works correctly, so this time it is an easy question to answer. Verifiability is the only important feature. It would be a more difficult question to answer if I had to worry about speed and space.

I bring the issue up because schools tend to emphasize efficiency, and as a result young programmers tend to do a bad job by optimizing too well. Suppose such a programmer is faced with the job of writing the Sort routine. First he wastes time calculating logs and powers to determine which is the optimum sort routine to use. Sort routines work for numbers, not playing cards, so he has to modify it to sort cards. (If it is generic this may be as simple as defining the < operator.) Then he has to verify it to see if it works. This is probably going to take longer than coding and verifying a simple sort routine. Time is money. The optimized version costs more (because it took more time to develop), and doesn't do the job any better. That's bad engineering.

Furthermore, the optimized version may cost more again later in the life cycle. If the program fails to work (or needs to be modified to sort by suits as well as ranks), a maintenance programmer will have to look at the code and figure out what it is doing. A simple sort routine is easier to verify than a complicated one. Therefore it will take less time (that is, cost less) to maintain the simple version than the optimized one.

4.6.14. Hidden Dependencies

In a moment, you are about to see the Draw_Poker program listing. Before you look at the whole listing, let me tell you that the first line is with PLAYING_CARDS, MONEY;. This makes it appear to depend only on two other software components, but you already know that MONEY depends on DIM_INT_32, ASCII_UTILITIES, and STANDARD_INTEGERS. DIM_INT_32 depends on STANDARD_INTEGERS and INTEGER_UNITS. ASCII_UTILITIES depends on STANDARD_INTEGERS. PLAYING_CARDS depends on RANDOM_NUMBERS (which you will see in the next section) and STANDARD_INTEGERS. RANDOM_NUMBERS depends on STANDARD_INTEGERS and CALENDAR. Who knows what the CALENDAR package body needs.

That's just a list of the units you get from the context clause. Some of the subunits of Draw_Poker depend on SCROLL_TERMINAL (to simulate hardware I/O), and would depend on special interface packages if the product was ever built. SCROLL_TERMINAL depends on VIRTUAL_TERMINAL, which may depend on DOS, VMS, or CURSOR and TTY. There's no telling what the special interface packages might need.

Usually we try to avoid hidden dependencies because they might cause unexpected side effects. We don't want to be unpleasantly surprised if we make a change to one module and find out that an apparently unrelated module doesn't work anymore.

The really amazing thing about Ada is that all these dependencies exist, but you don't have to worry about them. The dependencies are hidden in the sense that they don't clutter the program listings, but they aren't undocumented. Ada keeps track of the dependencies, so tools can be written that tell you all the units that will be affected if you make a change to a particular unit. Even if you don't use such a tool, Ada always makes sure that everything is current and that all the interfaces match before she will link object code modules into an executable image.

Ada uses layers of abstraction to hide these dependencies so they don't confuse you. You can obtain the power of so many previously written components with so little effort, and without cluttering your program with the details of how they work. The Draw_Poker program appears to be just a couple of pages, but it generates a sizable program because it takes such good advantage of reusable software components.

4.6.15. Building from the Bottom

I haven't said so, but the PLAYING_CARDS package is a bottom-up design. I tried to keep this a secret until now because bottom-up design is frowned upon in some circles. It earned a bad reputation because undisciplined programmers often start at the bottom of a design and keep building. When you start at multiple roots, you have to be incredibly lucky for all of these roots to grow together into a neat solid trunk. Normally there is a burl where things that don't really fit together have been forced into place. A pure bottom-up design usually isn't very good.

That doesn't mean all bottom-up designs are bad. I've shown you how you can build a SCROLL_TERMINAL on top of a VIRTUAL_TERMINAL that is built on top of operating-system interface packages. That's a bottom-up design, and it's good. Bottom-up design helps you write basic utility programs that can be used as building blocks in many different programs.

You only get into trouble when you try to get these individual building blocks to continue to grow and somehow merge with each other to form one program. You just can't start from many places and expect to be able to join them all with one golden spike ([6]). They probably aren't going to line up. The key to success is to recognize when you have built all the foundation modules you need, then stop working from the bottom-up.

4.6.16. Top-Down Design

When it comes time to establish the program flow, I think it is best to start from the top and work down. This means stating the solution to the problem in the most general terms, then defining those general terms with more specific terms until the solution is spelled out in complete detail. We saw an example of this in Chapter 3.5.9, and here is another example.

Listing 68 shows the top level of the Draw_Poker program. Don't think for a moment it was written sequentially. I used a screen-oriented editor and jumped all over that file. I started with a top level skeleton and put flesh on it. That is, it began like this:

```
procedure Draw_Poker is
begin
     loop
     (cursor)
     end loop;
end Draw_Poker;
```

I knew I wanted an infinite loop. I just had to decide what to put in the loop. I began with comments derived directly from the requirements.

```
-- Play as long as the user is willing to bet.
-- Shuffle before every deal.
-- Deal a hand to the player.
-- Show him what he has.
-- Let the player hold or draw each card.
-- Replace any cards he may have discarded.
-- Show him what he has now.
-- Pay him if he won.
```

[6] Note to international readers who might be unfamiliar with American history. Two companies were given the task of building the first transcontinental railroad. One started from the East, the other started from the West. When they met in the middle, the last two sections of track were joined by a golden spike.

Then, under each comment I wrote a few lines of code to do what the comment said. Usually I just invented a subprogram to do it. For example, under -- Show him what he has. I wrote put(PLAYERS_HAND, VALUE);. I knew I would need a routine that would display the cards in the hand, and display the value of the hand (NOTHING though ROYAL_FLUSH). At that point in the program development I didn't really care how it worked, just so long as it did work.

I realized I would need a function called Value_Of that would look at a hand and tell me if it contained a winning combination of cards. I considered the merits of simply passing the PLAYERS_HAND to the put routine and letting put call Value_Of instead of the main program calling Value_Of and passing the result to put. You can see I finally decided to do the latter. I did this partly because I wanted to avoid having both put and Payout call Value_Of (both need to know the value of the hand), and partly because I wanted to make it obvious at the top level that put was displaying the value of the hand. (If the procedure call was just put(PLAYERS_HAND); then it would not be obvious that put calls Value_Of and tells the player if he has a winning combination or not.)

This approach allowed me to partition the problem into five smaller problems. If I had five programmers working for me, I could have assigned one the job of writing a procedure gets the player's wager. I could let the second one write a function that determines the value of a hand. The other three programmers could work on procedures that display the hand and value, let the player discard, and drop the player's winnings loudly in a dish.

When I wrote the top level program I didn't worry about any declarations. I just compiled the program and got lots of error messages. Then, based on the error messages, I declared objects (STOCK, PLAYERS_HAND, WAGER, VALUE), the data type Values, and two library packages (PLAYING_CARDS, MONEY). I find that easier than trying to guess what declarations I will need before writing the code.

4.6.17. Renaming Declarations

Time out for a short comment on a technical point. Notice I have included the line function "="(LEFT, RIGHT : MONEY.Cents) return boolean renames MONEY."=";. I needed that because of the line exit when WAGER = MONEY.Type_Convert(0);. Let's talk about those two lines for a moment.

If I just wrote exit when WAGER = 0; I would get a type mismatch error. WAGER is a dimensioned quantity. It is a value expressed in Cents. The number 0 is a pure number with no units attached to it. It could represent dollars, francs, guilders, or pounds sterling. I happens that 0 cents equals 0 francs regardless of the current rate of exchange, but that's just a coincidence. I have to convert 0 to 0 Cents, and I can do that using the Type_Convert function in the MONEY package.

Having done that, I now have a problem with the = sign. The visible meanings for the = sign include comparisons of integers, real numbers, and Values, but not Cents. The function that compares Cents is in the MONEY package (inherited from DIM_INT_32). It isn't directly visible. I have three choices. First I can use MONEY;, which makes everything in MONEY visible. Second, I can use this awkward expression: exit when MONEY."="(WAGER,MONEY.Type_Convert(0)); (I'm sure you can see why I avoided that solution.) The third choice is to use a renaming declaration to make the operation visible. I used the third option partly because I wanted to include an example of renaming in this book. I have a slight preference for the first solution (especially if there are several operators that need to be seen), but if organizational programming guidelines prohibit USE clauses, the renaming technique is a simple way to comply.

4.6.18. Prototyping

No amount of planning will ever anticipate all the problems you will encounter, and the sooner you find out where the problem areas are, the better. As soon as you have established a top-level design, it is a good idea to write a prototype of that design. You can take any shortcut you like. Use a different language on a different computer if it will help you get the job done quicker. The important thing is to practice solving the problem once so you will learn things you need to learn to solve the problem for real.

If I were really going to build and sell the Draw_Poker machine, I would be looking at a significant hardware investment. I would pay engineers a lot of money to design and embedded computer, graphic displays, a mechanism that accepts coins and bills, the winnings dispenser, and the control panel containing the buttons the players push to deal and hold cards. Before I spend all that money, I want to be sure of the design.

What do I expect to learn from a prototype? If I knew that I wouldn't have to build the prototype. I usually learn things I never would have thought of in a million years. The Draw_Poker prototype was no exception.

The Draw_Poker top-level design defined five separately compiled subunits. The prototype was built by writing the simplest possible bodies for those subunits.

The first subunit is the procedure get that gets the WAGER from some special hardware that recognizes the values of coins and paper money. I can easily simulate this using the SCROLL_TERMINAL to ask the user how much he wants to be, converting the input string to a number of pennies, and returning the amount. When I wrote this module (Listing 69), I had to check for error conditions. Some of these error conditions couldn't happen in the real machine. The value of the U.S. dollar is less than it has been in the past, but it isn't negative yet. The real machine won't have to check for negative values of money, but it will have to check for the minimum and maximum bets. All of a sudden I realized, "I never specified what the machine should do if the player enters less than $1 or more than $999.99." I said it shouldn't accept those bets, but should it just spit the money back out without comment? Should it tell the user what he did wrong? If so, should I flash a light behind a red plastic lens that says, "BET WAS TOO SMALL", or should I display that message in big red letters on the screen? These are decisions that could affect the control panel or display screen, and I should make them now, before the hardware is designed and built.

The second subunit of Draw_Poker is Value_Of, shown in Listing 70. Unlike the other subunits, this one won't get thrown away when I build the real machine. Putting it in the prototype gives us an opportunity to start testing it early in the design phase. It turned out that Version 1.0 of this subunit failed to recognize ACE, TWO, THREE, FOUR, FIVE is a STRAIGHT. I discovered that while playing with the prototype. A rigorous testing program may have discovered that flaw, but then again, it might not. It always pays to have as all the experience you can with a product before you begin to sell it.

The third subunit is put (listing 71). I was surprised to learn that it ran too fast. Draw_Poker shuffled and dealt all five cards before I got my fingers off the keyboard, and put displayed them before I was ready to see them. I can't explain why, but that made me feel uncomfortable when I was playing the game. I guess I missed the thrill of seeing the first three cards turn up hearts, and wondering, "Will the last two also be hearts?" When I added a one-second delay in the display loop, it made it a much better game.

I also didn't like the fact that the cards I held were redealt to me. (If you compile and run the prototype, you will find that after you decide to hold or discard each card, all the cards disappear, and you are dealt a new hand. Some of the cards in that new hand are the cards you elected to hold.) I didn't fix that in the prototype because it was too much trouble, but I learned something important even though I didn't fix it. I now know that a graphic display of the playing cards will have to be able to erase individual cards, and slowly move new cards into the empty holes. I can tell that to the person designing the graphic display before the design is started. If I hadn't done the prototype, I probably wouldn't have thought of that, and I would have been unhappy with a display that showed the whole hand all at once. It probably would have been difficult, time consuming, and expensive to go back and modify the display routine.

The fourth and fifth subunits, Discard_From (Listing 72) and Payout (Listing 73) aren't particularly interesting or informative, but you need them if you want to play the game.

I wrote this prototype on my IBM PC AT clone. That's much more power than I need to do the job. When building the production units, I want to put in the cheapest computer that will do the job. How do I know what size computer to use? Can I get by with a single-board 8086 computer with 640 KB memory, or will I need 80286 with 4 MB?

Without the prototype, I'd just have to make a wild guess. The prototype doesn't tell me all the answers, but it helps me make a reasonable estimate. The Alsys compiler will let me generate object code for the 8086 or 80286. I can compile the prototype both ways to see what difference it makes.

The size of the real program won't be exactly the same size as the prototype. There are major differences, especially in the display routines, but at least I can tell a little bit about the program size from the prototype. I know exactly how big the Value_Of function will be. I know how few bytes are needed for the top-level procedure. I'll have to put some serious thought into how much the other routines will take, but I can make some assumptions and do some experiments. I won't be able to estimate the program to within a few bytes, certainly, but I should be able to tell if I will need extended memory or not. As I work on each subunit I can revise my memory estimate and check to make sure I'm not getting into trouble. If I am in trouble, the sooner I find out about it, the better.

4.6.19. Validation and Verification

Before we sell the product we have to validate and verify the program. This two-step certification process (1) assures that the program contains modules or statements that satisfy every stated requirement (and no unstated ones), and (2) verify that each module operates correctly. Early in the program development it doesn't make sense to try to verify that each module operates correctly because most of them haven't even been written yet, but is never too early to validate the design against the requirements.

If we validate the Draw_Poker program at this point in the development, we find some interesting things. It does everything it is required to do, but it also does some extra things. It has a default bet of $1. There isn't any requirement to do that. It also tells the player if he has a winning hand before he discards and cards. It lets the player end the program by entering a $0 bet. These are extra features not found in the requirements, and we have to address them somehow. We will have to (1) change the requirements, (2) change the design, or (3) ignore the problem for now.

These discrepancies crept in because the requirements are for a coin operated machine, but I built a prototype on a general purpose computer. The coin operated game should run forever, but I have to be able to stop the prototype program so I can use the computer for other things. It's a real nuisance to have to reboot the system every time I want to stop the prototype. I added the zero bet to give me an easy way to quit. I don't want to change the requirements, nor do I want to change the design of the prototype, so I'll ignore the problem for the moment. If I were going to continue with this example, I would add some comments to the prototype code to remind me to change the design for the coin operated version. (I'm ignoring the fact I could just as easily use CONTROL-C to quit, because I wanted an example of an instance when I might purposely violate the requirements in a prototype.)

The default bet makes no sense in a coin-operated game. The bet is whatever amount of money has been inserted. It was convenient for the prototype, but not really necessary, and it violates the requirements. I should take it out.

The early display of a winning hand was a side effect of using the same display routine before and after cards were discarded. In this case I decided to change the requirements because it makes the game more attractive (that is, easier to sell) to the player. (I could have also solved the problem by changing the design to match the requirements. This is easily done by assigning VALUE := NOTHING; before displaying the hand the first time.)

4.6.20. Integration

Things that work individually don't always work when you put them together. Or perhaps they work, but they don't work the way you expected them to. You never find these things out until you integrate the system. Many software development projects leave integration till the end. I believe in early integration. This is easy to do in Ada, if you have written a prototype.

Suppose we have built the Draw_Poker prototype and done the validation on it. We've made the changes eliminating the default bet and the zero bet. We can integrate modules as soon as they are finished.

If I were actually going to build and market the machine, I imagine the get procedure would be done first. I've seen dollar bill changers and candy machines, so I know devices that recognize the value of money exist. A little investigation would probably turn up a list of vendors who sell something I could use. I'd pick one and figure out how to connect it to a parallel I/O port. There are probably two handshaking signals. The first lets the device tell the computer that money has been entered. The second lets the computer acknowledge that it has read the amount and is ready for the device to accept more money.

The get routine has to monitor the input handshaking line and read the I/O port every time some money is inserted in the device. Then it can add that amount to a running total and use the other handshaking line to indicate it is ready for more money. It also has to monitor the DEAL button. When the user presses the DEAL button on the control panel, it passes the WAGER up to Draw_Poker and clears the total.

The get routine could be tested using a breadboard circuit. The money input device and the DEAL button could be mounted on the breadboard and wired to a connector that plugs into the parallel I/O port. A simple test routine could be written to make sure it works. The body of the program might look something like this

```
    SCROLL_TERMINAL.put_line("enter money");
    get(WAGER); -- routine under test
    SCROLL_TERMINAL.put_line(MONEY.Image(WAGER));
```

Just put in a known amount of money, press DEAL, and check the screen to see if it correctly tells you how much you entered. Do this as often as it takes you convince yourself that it is working correctly.

After you are convinced it works, link this real get procedure in place of the simulated get procedure you used in the Draw_Poker prototype. When you put some money in the machine and press DEAL, the prototype does what it always used to do. (The terminal screen shows you some cards, asks you which you want to keep, and pays you if you won.)

The disturbing thing is that it doesn't do anything while it is waiting for you to enter money. It doesn't prompt you, flash lights, or anything. It just sits there until you put some money in it. Someone walking by the machine doesn't even know it is on! Now your prototype has told you something else. There is a flaw in the design.

You have to decide what to do. You could add something to the get routine to make it flash a light behind a lens that says, "What's your bet?". If so, you need to add that light to the control panel.

On the other hand, the machine has a nice color display monitor. You may want get to call a graphic cartoon routine that tells people to step up and put money in the machine. That change may involve turning get into a task, adding a Come_On task, and using a timed call in a select statement to call Come_On if the player hasn't entered any money lately. We're talking about major changes here!

Using the prototype to integrate pieces of the solution can warn you of problems when it is still early enough to do something about it. You don't want to find out that you need a "What's your bet?" light after you have manufactured 20,000 control panels. You don't have time to start developing a lot of new graphic routines just before the final design review. I think it is vital to use a prototype program as an integration test bed early in development.

4.6.21. Maintenance Manual

In most cases, if you give a maintenance programmer a source-code listing and a pile of documents relating to the program, the only thing the maintenance programmer will read is the source code. That's because the source code is the only thing that counts and the only thing you can trust. It doesn't matter what it says in the documentation, the computer is going to do what the source code tells it to do. People have great intentions of keeping the documentation correct and up-to-date, but they seldom do. Often it is poorly written and confusing. Maintenance programmers usually don't read it. You may not like it, but those are the facts of life.

Since the only thing you can be sure the maintenance programmer will read is the source code, that's where the bulk of the information has to be. Chapter 2.11.1 described in detail how to document specifications and bodies. Putting these important comments in the source code (instead of a separate document) increases the probability that someone will read them.

Even if you write good comments in the source code, there are still things that need to go in a maintenance manual. The problem is getting the maintenance programmer to read the maintenance manual. Most people want to put everything in the maintenance manual. They want structure charts and data flow diagrams for every module. They wind up with a massive, expensive document that's boring and hard to read. All the information is there, but nobody can find it in the clutter. Few people have the patience to even try. That's why I believe it is important to keep the maintenance manual short and well organized. If you give someone a small, helpful document, he might read it.

A good maintenance manual begins with the theory of operation. This is a brief overview of what the program is doing. A few carefully chosen diagrams (structure charts, state transition diagrams, or data flow diagrams) should be used, but there is no need for diagrams of every module. After giving an overview, you should list the modules and tell how each module fits in the general scheme.

Analysis and design decisions should be documented in the body of the maintenance manual. If there were several viable ways to do something, explain why one approach was taken and others rejected. You should devote a subsection to each software component.

Section 3.1 of this book is a pretty good example of a maintenance manual for the ASCII_UTILITIES package. It gives general background information for the subprograms in the package, and Section 3.1.4 explains why I used the short- circuit control forms in some places but not others. These are the kinds of things that need to be documented but don't belong in the code itself. Section 3.5.9 could be turned into a maintenance manual for the Get_Response subunit by adding a structure chart and data flow diagram. Section 3.6 is NOT a very good example of a maintenance manual because it is too long and goes off on too many tangents. (It was written to be a tutorial of general concepts, not a maintenance manual for the FORM_TERMINAL. I was looking for excuses to digress and found lots of them.)

4.6.22. Other Software Engineering Concepts

There are three other things you need to consider when working on a big project that you don't need to worry about for small projects. They are (1) configuration management, (2) error reporting, and (3) cost and schedule. These things don't have a lot to do with Ada and probably don't belong in this book at all, but I just wanted to call your attention to them briefly.

4.6.22.1. Configuration Management.

A big part of software engineering is configuration management. When you build a large software product, you have to break it down into modules. The side effect of modularization is that you now have a lot of little pieces to keep track of. You have to know which pieces you need to build a larger unit. The little pieces are often revised, and you need to make sure you are using the correct revision.

Ada takes care of some aspects of configuration management. She knows what units need to be linked to create a main program. She knows if any of the units are obsolete. This could lead you to believe there is no need for configuration management if you use Ada. Unfortunately, that's not true.

Ada doesn't relieve you of the responsibility of configuration management. I've been trying to keep current copies of all the listings in this book on an AT clone (with the Meridian compiler), a genuine AT (with the Alsys compiler) and a VAX (with the DEC compiler). Every time I make an improvement in a listing on one machine I have to remember to make the same correction on the other two. It is a nightmare. Even if you use Ada, you still have to use some discipline, and/or a configuration management tool, to keep things straight.

4.6.22.2. Error Reporting.

Another important part of the life cycle is error reporting. This is perhaps just another aspect of configuration management because you need to keep track of the software errors you discover during development and after delivery. This should include a description of the symptoms, the consequences of the failure, the revision it was found in, and the correction to the software. I've never seen anything that makes me believe error reporting is any easier or more difficult in Ada than any other language.

4.6.22.3. Cost and Schedule.

For a little program like More, it would take you longer to estimate how long it will take than it takes to do it. Planning isn't an issue in these cases. Big programs involve massive expenditures over long periods of time, and you need to have a good idea of how much time and money the project will involve.

I always use time and money in the same breath, because for software development they are practically the same thing. If you want to estimate how much a software project will cost, it really comes down to estimating how manyman- hours it will take. There may be a few expenses that don't have anything to do with labor, but they are easy to estimate. You may have to buy a compiler or other software engineering tools, but you can pick up the telephone, call a few vendors, and you know what you will have to spend for them. The real trick is figuring out how many people you need and for how long, so you know how much you will have to spend in salaries and office expenses.

Other costs (such as testing, documentation, configuration management, and error reporting) will be related to the size of the project, which is related to how long it takes to develop. So the problem really boils down to "how do you know how many man-hours will be required to complete the project?" If you know how many man-hours it takes to write the software, you can multiply by some factor (that you have determined from your previous experience with software projects) to determine the overhead and support costs.

I'm still searching for a method that reliably predicts the duration of a software project. Experience seems to indicate that software development lasts as long as there is money to fund it, so if you tell me how much money you will give me, I will tell you how long it will take. I know that's not the answer you want to hear, so let me try another one. My first estimate is usually a wild guess. You probably don't like that answer any better, but that's the only honest one I can give you.

Although I don't have a good way to get an initial schedule estimate, I do know a way to tell if I am on schedule. After a little while, I can revise the estimated schedule based on progress made in the elapsed time.

How do you measure progress? When you've written 100 lines of code, are you 1% done or 10% done? You can't tell unless you know the program is going to be 1,000 lines or 10,000 lines, but you won't know that until the project is all over. Then it's too late. Counting lines of code doesn't tell you anything.

Fortunately it is a little easier to measure progress in Ada than other languages. That's because you can partition the work into modules and figure completion on the basis of number of modules completed. For example, take the Draw_Poker program. At the beginning of the program, you know Draw_Poker consists of the modules PLAYING_CARDS, MONEY, Get, Value_Of, Put, Discard_From, and Payout. There are seven major modules, so if you'll settle for a crude estimate you can figure that each module is one-seventh (14%) of the job. Each time a module is completed, you know you are another seventh of the way home.

To be more accurate requires a little more effort, judgment, and skill. Let's assume that MONEY is a completed reusable component and you don't need to figure it in the schedule. That leaves us with six modules. Let's rank them in order of difficulty. I think Put will be the hardest (it involves complicated graphics). The Get and Payout modules will be moderately hard (because they interface with hardware I don't have yet). PLAYING_CARDS is likely to be lengthy. Value_Of and Discard_From will be the easiest. If I let one unit of effort be the effort required to write the easiest module, I can estimate the relative difficulty of the others. PLAYING_CARDS, I feel, will take five times as long as Discard_From. Payout will probably take twice as long as PLAYING_CARDS, and so on. Intuition (and that's all it is) tells me the effort to complete the whole program can be allocated in this way.

Units of Effort	Module Name
50	Put
10	Get
10	Payout
5	PLAYING_CARDS
1	Value_Of
1	Discard_From

That all adds up to 77 units of work. Each unit of work is worth 1.3% of the total job. When Payout is done, then the project is 13% complete. When Put is done, it is 65% complete. If Payout and Put are both done, the job is 78% finished.

We don't have to wait for a module to be complete to estimate how far along we are. PLAYING_CARDS consists of eleven subprograms, which are probably equally difficult. If any three of the eleven are done, then 27% of PLAYING_CARDS is done. Since PLAYING_CARDS represents five of the seventy- seven units of work (6.5%), then 27% of 6.5% of the job is done. (It is 1.76% done.)

Once a month, I can measure my progress. If each month shows 5% increase in the amount of completion, I can revise my estimate to 20 months, regardless of what the initial wild guess was. (Total project time = time spent so far / fraction complete. 20 months = 1 month / 0.05 = 2 months / 0.10.)

It probably won't be nice and linear because you probably didn't estimate the relative difficulties of the individual software components properly. When you discover some part of the project is more difficult than expected, you can change the relative difficulty based on actual experience. As you get farther along in the project, the estimate should get better. If your initial wild guess was high, it will predict an early completion. If the wild guess was low, it will tell you that you are behind schedule in a few months.

4.7. Conclusion

We've taken the Draw_Poker program about as far as we can without buying some hardware and actually building it. In the process, we've talked about every aspect of writing large or small programs that I could think of, except one. The only thing we haven't talked about is how to test software. I've avoided that issue until now because it is a subject worthy of an entire chapter.

Chapter 5. TESTING SOFTWARE COMPONENTS AND PROGRAMS

Once you have written a program, how do you verify that it works as designed? The customary approach to verification is testing. Unfortunately, testing only shows the presence of errors, not the absence of them. If you test a program and don't find any errors, it doesn't necessarily mean there aren't any errors in it. It often means you didn't look hard enough. Testing can prove the absence of errors only when 100% of all possible conditions are tested. There are rare instances when 100% testing is feasible, but most of the time it isn't.

Generally, people test a program until they have confidence in it. This is a nebulous concept. Generally you will find many errors when you begin testing an individual module or collection of modules. The detected error rate drops as you continue testing and fixing bugs. Finally the error rate is low enough that you feel confident that you have caught all the major problems.

How you test your software depends on the situation. In this section we will look at some of the code developed in previous sections, and see some of the different things we can do to convince ourselves that it is correct.

5.1. Software Test Plans

Large projects usually test their product in accordance with a software test plan. Or, at least they SAY they do. The test plan is filled with "motherhood" statements saying that each module will be thoroughly tested, with special emphasis on values just inside and outside the nominal input limits, and values clearly outside the normal input range to stress it. It sounds great on paper, but it is usually impractical. I've never seen a software test plan that wasn't a waste of time.

Suppose you tried to test the Write program in accordance with such a software test plan. What file names are just inside or outside "nominal input limits?" Is "C:\HERECOMES\JAWS.NOW" more stressful than "BAMBI.AAH"? Suppose the file you try to write is an object code file rather than an ASCII text file. What constitutes success or failure?

I don't care what the software test plan says, testing is an Ad Hoc procedure. The incredible amount of time it takes to write, debate, and approve the test plan, (and especially the time it takes to explain why you didn't test in accordance with it), could be much better spent actually testing the software.

5.2. What is Truth?

We test algorithms to find out if they give the right answers. The problem is, how do we know what the right answers are? In this subsection we will look at several different ways of answering the question "What is truth?"

5.2.1. Trust

Suppose you want to test the Cos function in the TRIG package. You ask it to compute the cosine of 23 degrees, and it tells you the answer is 0.92050. Is that right? Granted it probably isn't exact, because the true value almost certainly needs more than five decimal places for an exact expression; but is 0.92050 the closest value that can be represented by five decimal places? How do you know what the right answer is?

Perhaps you have a book containing trig functions, and you compare the answer to the value in the table. How do you know the table is correct? It was probably generated by computer program. How do you know its computer program is more accurate than TRIG.Cos? You could check the result with your pocket calculator. Are you going to take the word of a $29 piece of plastic containing a 4-bit processor over a real 16- or 32-bit computer?

When it comes right down to it, it becomes a question of trust. You will accept the math tables in the handbook because it is published by John Wiley & Sons, and therefore can not contain any errors. Perhaps you trust the pocket calculator because it was made by a well known manufacturer with a reputation you can trust. Your confidence may be boosted by the fact that the math tables and the calculator both agree. For one reason or another, you put your faith in something.

Listing 74 shows the program I used to test the TRIG.Cos function with input values from -5.0 degrees to +370.0 degrees in 1 degree steps. That's almost 400 values, but it only takes the computer a few seconds to generate them all. (You can easily test more values by changing the loop increment to 0.1 degree or 0.001 degree if you like.)

The program creates a data file listing the 376 input angles and the function result of each angle on the same line. At 50 lines per page, that's almost 8 pages of data. You can quickly scan the data, looking at angles that are multiples of 30 or 45 degrees and verify that those angles give the expected results. Then you can look for obviously wild values. If you don't find any, that gives you some confidence, but it doesn't prove all the values are correct. What are we going to do?

The trick is to try to obtain the result two different ways and compare the results. If they agree, the answer is almost certainly correct. If they disagree, you need to find a third method to independently check the answer.

One of the reasons for using the identity Cos(THETA) = Sin(THETA+PI_OVER_TWO) for the Meridian and Alsys versions of the TRIG package was to use a different way computing the cosine than the DEC version of the TRIG package used.

The DEC version is the one I trust. Digital Equipment Corporation has supplied scientific run-time libraries to FORTRAN programs for years, and I'm sure they have discovered an outstanding, accurate, method for computing all the trig functions. The Cos function in the TRIG package I use on the DEC machine simply calls Cosd function in the VAX/VMS math library.

The THETA+PI_OVER_TWO method I used in the Meridian and Alsys versions is mathematically inferior because it introduces round-off error. Suppose I want to find the cosine of an angle THETA near -90 degrees. Perhaps the angle THETA is exactly -1.57 radians. The cosine of that angle is 7.9632686e-4 (according to my pocket calculator). Suppose the value I use to approximate PI_OVER_TWO is 1.5708 radians. When I compute Sin(-1.57 + 1.5708) I get 7.9999986e-4, an error of 3.6729994e-6! That bothers mathematicians. It only bothers engineers if they know the angle THETA was measured to an accuracy better than 3.673 microradians (about 0.00021 degrees).

I ran the Cos_Test program on Meridian, Alsys, and DEC machines, expecting to get slightly different results. I wanted to know the magnitude of the difference. I also wanted to know what input angle gives the poorest results.

Cos_Test always creates an output file called COS.DAT. I could have modified the source code every time I moved the Cos_Test from one computer to another, or I could have made the program ask the user what file to put the output data in, but I decided it simpler to let it put the data in COS.DAT and use an operating system command to rename it to DECCOS.DAT, MERCOS.DAT, or ALSCOS.DAT after it was created.

Figure 40 shows some of the results I got with the Meridian version of the cosine routine. Eight pages of cosine data isn't very interesting reading, so I've shown a few segments of the output just to give you a sample of what it looks like. The DEC and Alsys results are almost identical, often differing by 1 in the least significant digit.

If a person compares two eight page printouts, and doesn't notice any major errors, that gives you some measure of confidence. People get careless sometimes, so you can't really be sure the printouts match. You could have more confidence if a computer meticulously compared both listings, line by line.

```
Figure 40. Portions of Meridian COS.DAT.
-------------------------------------------------
-5.00              0.996195
-4.00              0.997564
-3.00              0.998630
-2.00              0.999391
-1.00              0.999848
0.00               1.000000
1.00               0.999848
2.00               0.999391

29.00              0.874620
30.00              0.866026
31.00              0.857168

44.00              0.719340
45.00              0.707107
46.00              0.694659

59.00              0.515039
60.00              0.500000
61.00              0.484810

88.00              0.034900
89.00              0.017453
90.00              0.000001
91.00             -0.017452
92.00             -0.034899

134.00            -0.694658
135.00            -0.707106
136.00            -0.719339

178.00            -0.999391
179.00            -0.999848
180.00            -1.000000
181.00            -0.999848
182.00            -0.999391

268.00            -0.034901
269.00            -0.017454
270.00            -0.000001
271.00             0.017451
272.00             0.034898

358.00             0.999391
359.00             0.999848
360.00             1.000000
361.00             0.999848
362.00             0.999391
363.00             0.998630
364.00             0.997564
365.00             0.996195
366.00             0.994522
367.00             0.992546
368.00             0.990268
369.00             0.987689
```

MS DOS has a file comparison utility that I hoped to used to compare them. When I tried it I found it wasn't totally successful. So many lines differed by one digit that the utility program got discouraged and gave up. That meant I had to write a program to read two files and find the differences. I could have made the program compute the average difference and standard deviation, but I was more interested in the maximum error.

The COS_DIF program is shown in Listing 75. It asks the user for two file names and compares them, line by line. The first entry on each line (the angle tested) should be identical. If it isn't, then one of the files has been corrupted. The program ends with an error message telling where the error was detected. Otherwise, it goes through the whole file, keeping track of the maximum errors in the positive and negative directions. When it is finished it prints the maximum differences. When I used it to compare the Meridian results to the DEC "truth", I got the results shown in Figure 41. Results comparing Alsys to DEC outputs were similar.

Now I have confidence in the Meridian and Alsys versions of the TRIG.COS function, because I have compared them to a standard I can trust (a routine in the DEC math library).

5.2.2. Inverse Functions

Another way to discover What is truth? is to use inverse functions. I use this technique often. I tested the TRIG.Log function by taking the log of a number, then taking the antilogarithm of the result to see if I got the same number back. I tested the TRIG.Sqrt by squaring a number, taking the square root of the result, and comparing it to the original number. I tested the Fixed_Image, Float_Image, and Value functions by taking the image of a floating point number, and then finding the value of the image, and comparing the result with the original number.

In a perfect world you would always get exactly what you started with. Unfortunately round-off errors give you a result that is nearly equal to the original value, which makes comparison more difficult. You can take two approaches to when comparing the results. (1) You can establish an acceptable error, and declare any difference between the original value and the reconstructed value smaller than this threshold value to be OK. (2) You can measure the difference between the original and reconstructed values, and keep track of the maximum positive and negative errors.

The first method is most useful for situations where you know how accurate you need to be. For example, suppose the COORDINATES package is going to be used in a program that guides a missile to a target. If the missile warhead has a lethal radius of X feet, then the COORDINATES package must not introduce any inaccuracy resulting in a miss distance greater than X feet. It would probably be a good idea to be conservative, and set the threshold at X/2 feet, or X/10 feet, or whatever is appropriate to the application.

The second method is better for situations where you want to know what is the best you can do. For example, you could use the second method to find the maximum error introduced by the COORDINATES package. If the calculations can be off by as much as Y feet, then you will have to design the warhead to have a lethal radius of at least Y feet. It would probably be a good idea to be conservative, and make it 2Y feet or 10Y feet.

Both methods have weakness, especially in situations where there is one really wild point and many points that just barely fail. Suppose there is one input condition that gives absolutely crazy results (perhaps a

```
Figure 41. Accuracy of Meridian TRIG.Cos function.
-------------------------------------------------------
C:>cos_dif
What is the name of the REFERENCE file? deccos.dat
What is the name of the TEST file? mercos.dat
The maximum positive error was 0.000001 at 21.00 degrees.
The maximum negative error was -0.000002 at 308.00 degrees.
```

location near the origin where a division by a value near 0 occurs), and several other points that are moderately over threshold. The first method will tell you many points failed, but won't tell you the magnitude of the one spectacular failure. You might look at four or five points, see they are all just barely out of tolerance, and decide it's close enough. The second method will tell you the magnitude of the failure at that one awful point, but not tell you that there were a dozen other points that were 20% over the threshold. You might look at the one wild point, decide it is a pathological case that can't occur in normal operation, and think everything is OK.

The even functions, such as square root, are partially symmetrical. You can square 2 to get 4 and take the square root of 4 to get 2 again, but the method doesn't work when the original value is -2. This doesn't mean you can't use the method on partially symmetrical functions, you just have to be more careful. When testing the square root, for example, you could test it with positive values expecting to get the original value back. Then test a second time with all negative values, expecting to get the absolute value back. When testing trigonometric functions, you may want to test one quadrant at a time because inverse functions return mirror images of the input angle in certain quadrants. (The Arcsine of the Sine of 100 degrees is 80 degrees).

Inverse testing may leave some gaps in the test suite. The square root test described above won't tell you what will happen if you try to take the square root of a negative number because all the input values were created by squaring a positive or negative number, the result of which can never be negative. Furthermore, if you are testing a 32-bit integer square root, the test cases will be sparse at the higher values. If you square 46,339 it will yield the test input 2,147,302,921. Squaring 46,340 results in 2,147,395,600. There are 92,679 values between 2,147,302,921 and 2,147,395,600 that can never be tested using this approach.

5.2.3. Manufactured Data

Embedded computers are often used in applications where the input signals are corrupted by noise. An algorithm must separate the true signal from the noise. Again the question is, What is truth?

This sort of situation doesn't lend itself to the inverse function testing we just discussed. You can put noisy data in the input of a filter and get clean data out, but you usually can't shove the clean data in the output of the filter to reproduce the same noisy input.

If you have an established algorithm that is believed to work, you can put faith in its results, using the first method we discussed. You just record some noisy, real-world data, process it with both algorithms, and compare the results. But if you are developing an entirely new algorithm, then you don't have an old one to tell you what the right answer is. If you do have an old one, and it gives different results than the new one, then it might test your faith. Are you really sure the old algorithm gives the right answers? There is likely to be some doubt in your mind.

In situations like these I like to use a different approach. I manufacture some realistic input data by starting with clean data and adding artificially generated, known noise. Then when I use an algorithm to separate the signal from the noise, I know exactly what the correct answer should be.

The description of a valid input signal for your algorithm depends entirely on the problem, but generally you can create a pure, valid input signal by adding several sine waves with various amplitudes, frequencies, and phase differences. You can compute the value of this input signal at the moments of interest and put it in a file called CLEAN.DAT. Use this as the input to your algorithm and store the resulting output in TRUTH.DAT.

The next step is to create some realistic corruption. Maybe the signal is corrupted by frequencies that are out of band, or periodic noise spikes. If the signal is likely to be corrupted by gaussian noise, you can use the RANDOM_NUMBERS.Noise function to create a noise waveform. If appropriate you can add a bias and/or filter the noise waveform. You should have a good idea of the characteristics of the noise in your system (if you don't, you haven't done your homework), and should be able to model it. Simulate a representative noise waveform and store it in NOISE.DAT.

The third step is to add each element in CLEAN.DAT to the corresponding element in NOISE.DAT, and store the result in INPUT.DAT. You now have a realistic, noisy input signal for testing your algorithm.

Finally, process the noisy input and store the result in OUTPUT.DAT. You can compare OUTPUT.DAT to TRUTH.DAT to see how well the algorithm performed in the presence of noise. You can do this as often as you want with different kinds of input signals and different kinds of noise.

5.3. Selecting Input Conditions

Now that we know what truth is, we need to know how to select input values. The five methods generally used to do this are 100% testing, uniform sampling, Monte Carlo testing, good judgment, and dumb luck. I hesitated to include the last method, but lets be honest-- that's how many error are found!

5.3.1. Testing Every Case

Clearly the best thing you can do is to test every possible input. This usually isn't practical, but in some special cases it is. For example, all possible inputs to the ASCII_UTILITIES.Upper_Case function that returns a character can be easily be tested. Just write a program that loops through all 128 ASCII characters and compares the input character with the output character. If they are the same, do nothing. If they are different, then print the input character, the output character, and call new_line. You should get 26 lines of output that look like this:

```
aA
bB
cC
(and so on)
xX
yY
zZ
```

Here's another case where 100% testing is possible: Suppose we want to test the COORDINATES package, and the input values are given as one byte integers. That means the values can range from -128 to +127 feet. Figure 42 shows a short test program to check the COORDINATES package transformations for all possible inputs. It uses the inverse function technique to test both Transform functions by transforming a rectangular point into a polar point, and then transforming it back again. The program does this for all rectangular positions included in the area from 128 feet west to 127 feet east, and 128 feet south to 127 feet north. It keeps track of the largest errors, and prints them when finished. It also tells us how long it took the test program to run.

The two previous examples are exceptions to the general rule. It is more likely that the COORDINATES package has to work for all locations 100 miles from the origin. In theory, we could 100% test this package by simply changing the loop limits to +/- 528,000 feet to get full 100 mile testing coverage. In practice we can't do this. It took 177 seconds for the test program shown in Figure 42 to run on a 10 MHz AT Clone. The number of points tested was 256**2 = 65,536 points. That's 370 points per second. To test (2*528,000 + 1)**2 points would take 753,473,124 seconds. That's almost 24 years. By the time we finished testing it, the weapon it was being designed for would be obsolete.

```
Figure 42. Coordinates_Test program.
-------------------------------------------------------
with COORDINATES, CALENDAR;
with STANDARD_INTEGERS; use STANDARD_INTEGERS;
with TEXT_IO; use TEXT_IO;
procedure Coordinates_Test is
  use COORDINATES; -- for Type_Convert and "+"
  R, WORST_MAX_NORTH, WORST_MIN_NORTH,
  WORST_MAX_EAST, WORST_MIN_EAST
    : COORDINATES.Rectangular_points;
  P : COORDINATES.Polar_points;
  EAST, NORTH, ERROR : COORDINATES.Feet;
  MAX_NORTH_ERROR, MAX_EAST_ERROR
    : COORDINATES.Feet := +(Integer_32'FIRST);
  MIN_NORTH_ERROR, MIN_EAST_ERROR
    : COORDINATES.Feet := +(Integer_32'LAST);
  START_TIME, STOP_TIME : CALENDAR.Day_Duration;
begin
  put_line("Starting COORDINATES Test");
  START_TIME := CALENDAR.Seconds(CALENDAR.Clock);
  for i in Integer_32 range -128..127 loop
    EAST := Type_Convert(i);
    for j in Integer_32 range -128..127 loop
      NORTH := Type_Convert(j);
      R.NORTH := NORTH;
      R.EAST  := EAST;
      P := COORDINATES.Transform(R);
      R := COORDINATES.Transform(P);
      ERROR := R.NORTH-NORTH;
      if ERROR > MAX_NORTH_ERROR then
        MAX_NORTH_ERROR := ERROR;
        WORST_MAX_NORTH.NORTH := NORTH;
        WORST_MAX_NORTH.EAST  := EAST;
      end if;
      if ERROR < MIN_NORTH_ERROR then
        MIN_NORTH_ERROR := ERROR;
        WORST_MIN_NORTH.NORTH := NORTH;
        WORST_MIN_NORTH.EAST  := EAST;
      end if;
      ERROR := R.EAST-EAST;
      if ERROR > MAX_EAST_ERROR then
        MAX_EAST_ERROR := ERROR;
        WORST_MAX_EAST.NORTH := NORTH;
        WORST_MAX_EAST.EAST  := EAST;
      end if;
      if ERROR < MIN_EAST_ERROR then
        MIN_EAST_ERROR := ERROR;
        WORST_MIN_EAST.NORTH := NORTH;
        WORST_MIN_EAST.EAST  := EAST;
      end if;
    end loop;
  end loop;
  STOP_TIME := CALENDAR.Seconds(CALENDAR.Clock);
  put("The computed value for NORTH was");
  put(Integer_32'IMAGE(Dimensionless(MAX_NORTH_ERROR)));
  put(" HIGH at");
  put("(" & Integer_32'IMAGE(
    Dimensionless(WORST_MAX_NORTH.NORTH)));
  put("," & Integer_32'IMAGE(
    Dimensionless(WORST_MAX_NORTH.EAST)));
  put_line(").");
  put("The computed value for NORTH was");
  put(Integer_32'IMAGE(Dimensionless(MIN_NORTH_ERROR)));
  put(" LOW  at");
  put("(" & Integer_32'IMAGE(
    Dimensionless(WORST_MIN_NORTH.NORTH)));
  put("," & Integer_32'IMAGE(
    Dimensionless(WORST_MIN_NORTH.EAST)));
  put_line(").");
  put("The computed value for EAST was");
  put(Integer_32'IMAGE(Dimensionless(MAX_EAST_ERROR)));
  put(" HIGH at");
  put("(" & Integer_32'IMAGE(
    Dimensionless(WORST_MAX_EAST.NORTH)));
  put("," & Integer_32'IMAGE(
    Dimensionless(WORST_MAX_EAST.EAST)));
  put_line(").");
  put("The computed value for EAST was");
  put(Integer_32'IMAGE(Dimensionless(MIN_EAST_ERROR)));
  put(" LOW  at");
  put("(" & Integer_32'IMAGE(
    Dimensionless(WORST_MIN_EAST.NORTH)));
  put("," & Integer_32'IMAGE(
    Dimensionless(WORST_MIN_EAST.EAST)));
  put_line(").");
  new_line;
  put("The test took");
  put(integer'IMAGE(integer(STOP_TIME-START_TIME)));
  put_line(" seconds to run.");
  new_line;
  put_line("Done.");
end Coordinates_Test;
```

5.3.2. Sparse Uniform Testing

Sparse Uniform Testing is often a viable alternative to 100% testing. We could start a test program running at 4 P.M. Friday afternoon and check the results 7 A.M. Monday morning. That means the test program can run 63 hours. At 370 points per second the Coordinates_Test program could check 83,916,000 points. That's about 9160 squared, so we could let the two loop indices go from - 4580 feet to +4580 over the weekend. If we wanted to check the algorithm to make sure it works over a 100 mile range we could let EAST and NORTH be 115 times the loop index. The program could run over the weekend and we would have uniform test coverage over the whole area.

The coverage wouldn't be as dense as 100% testing because we would be checking it every 115 feet instead of every foot, but this kind of sparse uniform testing is good because it quickly covers the entire range of inputs and is likely to uncover overflow conditions. (In fact, 46,341 squared overflows on a 32-bit integer machine, so the transformations fail at 8.78 miles.)

We said it would take almost 24 years to test all the points in a square +/- 100 miles from the origin (to 1 foot resolution), but if you add a third dimension and want to test all those points at all altitudes from 0 to 8 miles in 1 foot increments, the schedule out stretches out 971,520 years. Nobody is going to fund a project that long!

We've already calculated that the test program can test 83,916,000 points over the weekend. If we uniformly distribute those test points over the area 8 miles high, 100 miles each direction from the origin, then the points fall on grid lines 825 feet apart. That's sparse, but it gives gives good, uniform coverage over the entire area.

The disadvantage of uniform testing is that the inputs are regular multiples of the index. Therefore certain ratios of input conditions occur over and over, and other ratios never happen. Differences between two variables tend to be multiples of certain values. This nice, regular input pattern sometimes misses small differences of large numbers, or division by numbers near zero.

Another problem with uniform testing is that it is too uniform. It spends just as much time testing the coordinate transformations of targets 90 miles away as it does testing coordinate transforms 90 feet away.

If you are testing an early warning system you want to test lots of cases at the outer boundary, and don't care about short ranges. If you are testing a short-range gun system, you care more about the performance at short range than long range. In these cases you don't want a uniform distribution. Monte Carlo testing might be more appropriate.

5.3.3. Monte Carlo Testing

Real data often isn't regular and predictable, so it is sometimes a good idea to test algorithms with pseudorandom data. This is especially true when there are more than just two variables and when sparse uniform testing is awfully sparse. Monte Carlo testing is an important part of software engineering.

Let's suppose we decided to test the COORDINATES package using Monte Carlo techniques for picking the X and Y coordinates, instead of using a uniform distribution. We still have all weekend, so we want to generate about 84 million coordinate pairs and test them. If we try to do that, we will be in for several surprises.

First, we will find the program is still running monday morning when we get to work. That's because it takes more time to generate a random number than it does to simply bump the index in a loop. You won't be able to test as many points in a given time period as you will if you use a uniform distribution.

Second, you may not be testing as much as you think. Suppose you generated X and Y by doing this:

```
    X := integer(RANDOM_NUMBERS.Rnd * (2.0 * 528000.0) - 528000.0);
    Y := integer(RANDOM_NUMBERS.Rnd * (2.0 * 528000.0) - 528000.0);
```

It appears that you are generating a random distance from 0 to 200 miles (in feet) and subtracting 100 miles (in feet) to get a random distance from -100 to +100 miles (expressed as a number of feet). Well, you are, but the distances aren't as random as you expect.

5.3.3.1. RANDOM_NUMBERS package.

You've read a little bit about the RANDOM_NUMBERS package already. It was used in the PLAYING_CARDS package to deal the cards. It was also used to create noise for manufactured data. I didn't say much about it then because it wasn't important to know how it works. Now we need to understand its limitations, so lets look at Listings 76 and 77.

If you read the fine print in the RANDOM_NUMBERS package specification, you will see the Rnd function generates a random sequence that repeats after 2048 numbers. After you have generated 1024 X,Y pairs, you will begin to repeat the same pairs. So the algorithm above doesn't really test 84 million random positions. It tests 1024 random positions 82,000 times. The last 81,999 times don't tell you anything you didn't already learn the first time.

Furthermore, every random number generated by Rnd can be expressed as N/2048 where N is 0 through 2047. If you multiply by 2*528000, then every value can be expressed as 515.63 * N, where N is an integer from 0 through 2047. So even though it appeared to be testing 84 million different points where X and Y could take on any integer value of feet, it was really only testing 1024 sparsely distributed points.

The moral of the story is, "You better know the characteristics of your random distribution, or you could badly mislead yourself."

If you read the whole RANDOM_NUMBERS package specification, you will see a procedure called Random_Digit, that returns a random integer in the range 0..9. The comments in the package body describe how it works. I don't think the sequence repeats, but I haven't proved it. (Proving a random sequence doesn't repeat, and is completely uncorrelated is quite a job.) If you need a sequence longer than

2048 numbers, or need a very dense sequence, then use the Random_Digit function to build random numbers one digit at a time. Be patient, though. The Random_Digit function is slow.

Let me be the first to admit this isn't a very good random number package. Remember, this is a book about how to write Ada programs, not about how to design the ultimate random number generator. I didn't want to get waste a lot of time confusing readers with things that don't really have anything to do with Ada. There must be people who have made it their life's work to figure out the fastest, longest, densest random sequence. If you need a really good random number generator, it would be a good idea for you to search the literature and rewrite the package body of RANDOM_NUMBERS using a better technique. I just needed something that would generate a random sequence of 52 numbers so I could shuffle the poker deck, so I used this simple, well known random number generator. It's plenty good for that purpose.

Despite the limitations of the simple-minded random number generator, it teaches a valuable lesson: "Simple approaches can be taken in a package body when you need to do a quick feasibility study, then upgraded later." Maybe the RANDOM_NUMBERS package isn't good enough for a real Draw_Poker program. That doesn't matter when I'm first testing the logic of the program. Who cares if somebody notices the 157th card after the three of clubs is always the eight of hearts? Before I start selling the machine, I can give the RANDOM_NUMBERS package specification to somebody who really enjoys writing random number generators. When they come back with an improved body, I can just compile it and link it, and I will not have to worry about making any other changes in the Draw_Poker program. The important thing is that I can test out the Draw_Poker concept now, using a mediocre random-number generator.

Starting with the simplest approach helps me define the requirements for the final solution. For example, I learned that I have to throw in 1 second delays just to slow the Draw_Poker program down. This means I can tell a designer to come up with a random number generator without any detectable correlation (that would give a gambler an edge) and no speed constraints. On the other hand, if I need the random number generator for testing the COORDINATES package, and find out the simple package body only lets me test 10 points a second, speed will be important to me. I'd accept something that quickly generates a long string of dense random numbers derived from the CPU clock, even though it has a mild (but noticeable) 60 Hertz correlation in it.

5.3.4. Good Judgment

When you do Monte Carlo testing, you have to know what you are doing. Let's accept that fact. When I hear people promoting their software development methodologies, they tend to claim their way is so simple it doesn't even require thought. They say you could train monkeys to follow the rules, and they could do it for you. This line is attractive to managers who want to hire hoards of people for minimum wage. The truth is, there is no substitute for intelligence and good judgment.

Testing the ASCII_UTILITIES.Upper_Case function that works on a whole string is an example of a situation where all you can use is good judgment when selecting test cases. You can't test 100% of all possible input strings, simply because you can't even list 100% of all possible input strings. Even a sparse uniform distribution of all possible input strings boggles my mind. It would be possible to use Monte Carlo techniques for generating random input strings, but how many would I need to generate to assure myself it works? If there are thousands of these strings generated, how do I check to see if they were properly converted by the Upper_Case function? This is a case where, like it or not, there is no substitute for intelligence. You have to use some good judgment and come up with a few, well-chosen test cases.

It's hard to teach good judgment. Techniques that work one time don't make any sense at other times. I'll just give you an example, and then I'm afraid you're pretty much on your own.

The string Upper_Case function has three basic parts: (1) There is a loop that indexes through all the characters in the string, (2) the assignment statement that gives values to each character in the string, and (3) the return statement. I need to convince myself that each of these three parts work, then I have some assurance that the whole thing works.

To test the loop index, I will need to try the function on strings of different lengths. I certainly want to try one-character strings, and some moderate length strings, but what do I do about bizarre string lengths? How do I test it for a string -2 characters long? How do I declare the input test string? Will Ada let me declare TEST : string(2..1);? What value will she let me assign to TEST? What exception do I expect Upper_Case to raise if I do succeed in giving it an invalid string? What do I define the proper response to be?

Of course I have to ask the same questions about strings that are longer than the maximum length. Those questions are a little more plausible because it isn't unreasonable to anticipate a line like X := Upper_Case(Y & Z); where the lengths of Y and Z exceed the maximum string length. But even in that case I have to declare X to be longer than the maximum length, and Ada probably won't let me compile my test case. If she does, she should raise CONSTRAINT_ERROR when she tries to elaborate X, and the program will terminate before I get to my test case.

So here is my stand: Despite all my best efforts, I can't consciously create a situation that tries to process an illegal string, so I probably won't create one by accident. If I do create one by accident, it is the result of an error, and I should detect that error before I call the Upper_Case function. If I fail to detect the error, the part of the program I should fix is the part that caused the error or failed to detect it, not the innocent Upper_Case function. Judgment tells me that I don't need to test strings less than one character long, or longer than the maximum allowed by the implementation.

Therefore, to assure myself that the loop in the Upper_Case function works, I will test it with some minimum length strings, some maximum length strings, and a few intermediate length strings, and I will be satisfied if all those cases work.

Testing the assignment statement is relatively easy. I've already done 100% testing on the character Upper_Case function, so I know it works. Therefore, the few tests cases that checked the loop will also suffice to check the assignment. I'll want some of those cases to include lower case letters, and some to include upper case letters, numbers, and punctuation marks, to show they aren't changed.

It isn't hard to convince myself that the return statement works. The few test cases I've already planned won't work unless the return statement words, so they provide valuable information about the return statement. The only feature I need to test that hasn't already been tested is to see what happens when I try to Upper_Case a string N characters long and assign it to a string M characters long, when N is not equal to M. It should raise CONSTRAINT_ERROR. If N and M are known at compiler time, Ada may warn me about the CONSTRAINT_ERROR before I run the test.

Therefore, good judgment tells me that a few test cases satisfying the requirements in the preceding paragraphs are sufficient to test the string Upper_Case function.

5.3.5. Dumb Luck

Most people won't admit it, but a lot of errors are found by dumb luck. That's how I found a bug in Fixed_Image function.

The Fixed_Image function is one of those routines that is impossible to test for all possible inputs. There are an infinite number of floating point values. The computer can only represent a finite number of them, but even so, that finite number is far too large to test. Besides, the Fixed_Image function has three independent options (the number of digits before the decimal point, number after the decimal point, and leading characters). You can't test all input values for all combinations of options.

I had no choice but to use good judgment when selecting the test cases. I tested Fixed_Image with a bunch of different values. I used big values, small values, positive values, negative values, fixed length, variable length, and so on. It passed.

Then I wrote the some routines to test the logarithm functions in the TRIG package using the inverse function method. I was taking logs of powers of two, and then taking the antilog of the result to see if I got the original value back again. For example, Ln(0.5) should be -0.69315, and Exp(-0.69315) should be 0.5 (or very close to it). Ln(2.0) should be +0.69315, and Exp(+0.69315) should be 2.0. The test program was checking the results automatically, so there really wasn't any need to print them out, but I did anyway.

My test routine told me everything was fine, but I happened to notice on the printout that Ln(0.5) was +0.69315, and Exp(+0.69315) was 0.5. That's clearly an error in both the Ln and Exp functions. I thought they didn't work for values less than one. But then I noticed Ln(0.25) was -1.38629, and Exp(-1.38629) was 0.25, so the functions appeared to work for numbers less than one after all. Then I noticed Ln(2.0) was also +0.69315, and Exp(+0.69315) was 2.0! How could Exp(+0.69315) evaluate to two different answers and always the right one? It was very confusing.

It turned out that the bug was in Fixed_Image. Numbers in the interval from -1.0 to 0.0 were printed as positive values. Even though I had tested Fixed_Image with many different values, I didn't happen to pick a value in that small interval. I just discovered it through dumb luck.

After you have tried one or more of the previous input selection methods, dumb luck will finish the job. Just put a fully-debugged, error-free module to normal use and you will almost always find errors you didn't find before, no matter how much you tested it.

Don't think that dumb luck is a substitute for the other kinds of testing. It is the final line of defense, and should just find minor coding errors that can be fixed without changing any documentation. If normal usage finds an error that requires a major program revision, it is too late in the development cycle for that. You have to do everything you can to find major errors early.

Routines that involve user interfaces are very difficult to test, because it is so difficult to predict what a user will do. These routines need lots of operational testing because dumb luck is the only way of finding strange responses to illegal inputs.

5.4. Testing Mechanisms

After you have decided what method you are going to use to select the inputs for your test cases, and what outputs are correct, you are still faced with the problem of writing the code that will actually test your software. There are several ways to do this.

5.4.1. Test Drivers

A common way to test lower level modules is to use a test driver program. You've already seen the technique used in the Cos_Test program (Listing 74) and the Coordinates_Test program (Figure 42).

The approach is to write a program that passes input data to the unit under test and compares the output to truth. The input comes from a loop index for 100% or uniform testing, from a random number generator from Monte Carlo testing, from carefully selected input conditions coded directly in the program or from a file of manufactured data. The output is determined to be good or bad by comparing it to some trusted results, using an inverse function, or comparing it to a file containing data that is defined to be correct.

5.4.1.1. White-Box Testing.

Test drivers are good for testing a single unit in isolation, but can also be used to test several modules at once to save time if you do "white- box" testing. White-box testing takes advantage of things you know about the internal workings of the module under test.

You may have wondered why I tested the Cosine function that uses degrees instead of the Sine function that uses radians. I did that simply because it tests several features of the Meridian TRIG package at once. The Sin and Units_Convert functions are nested in the Cos function. As the index goes from -5.0 degrees to 370.0 degrees, it uses the Units_Convert function to convert from degrees to radians at each of those angles, so there is no need for me to waste time separately testing the Units_Convert function. I know the Cos function in the Meridian TRIG body calls the Sin function, so the Sin function is tested at the same time.

White-box testing saves time, but it has its disadvantages, too. If you are testing several things at once, and you get the right answer, it is a pretty good indication that everything works. If you don't get the right answer, then you can't be sure what went wrong. If the Cos_Test results are wrong, it could be because there is an error in the Cos function, the Sin function, or the Units_Convert function. Then you will have to test Sin and Units_Convert separately, using a similar test driver program, to isolate the error.

5.4.1.2. Black-Box Testing.

You have to know all about the internal workings of the module under test to do white-box testing. Since I know the Meridian Cos function calls Units_Convert and Sin, I took advantage of that fact to reduce the number of tests. The DEC trig functions are black boxes. Since I don't know how the DEC Cosd function works, I can't assume that Cosd calls Sind, Sin or Cos. (It probably doesn't.) If Cosd passes its test, it is still necessary to test Sin.

5.4.1.3. Which Color is Better?

Is black-box testing better than white-box testing? That's hard to say. Both methods have their advantages and disadvantages. You could argue that black-box testing is more reliable because you make fewer assumptions. For example, the white-box test of TRIG.Cos assumes that Cos always calls Sin. If the TRIG body is changed to call Cos directly, then Sin never gets tested.

On the other hand, you could argue that white-box testing is more reliable because you know more about the thing you are testing, and will be more alert to potential problems. For example, if you are white-box testing a Tan function, and know that it computes Sin/Cos, you may be more likely to remember to check for conditions that may cause a division by zero than you would if it was just a black box.

It really comes down to the person doing the testing, rather than the method used. Either method, consciously applied, will do the job. Either method, poorly done, will fail to catch errors.

5.4.2. Test Stubs

Since test drivers are higher-level routines that call lower-level subprograms, that makes it nearly impossible to use them for testing the top level of a program. (I say "nearly impossible" because there are situations where you can create a super-level driver that is one level higher that the top level.) In general, it is easier to test the higher levels of your program using a test stub instead.

A stub is a simple routine that takes the place of the real routine. It may be a null procedure, or it may simply write a message to the screen that says, "I was called!" These stubs let you check the interfaces and higher levels of the program.

Figure 43. Get_Command_Line stub 1.

```
-----------------------------------------------------------
--             GCLS1.ada
--             9 June 1987

--             Do-While Jones
--             324 Traci Lane
--             Ridgecrest, CA 93555
--             (619) 375-4607

-- Get_Command_Line, Stub 1


procedure Get_Command_Line(TAIL   : out string;
                           LENGTH : out natural) is
begin
  TAIL(1..8) := "SHOW.ADA";
  LENGTH := 8;
end Get_Command_Line;
```

When I was working on the Show program, I wanted to test the concept of adding the command-line input to the main program before I had figured out how to make the command line work. I did this using the test stubs shown in Figures 43 and 44. Figure 43 just returned a constant string. When I verified that it worked, then I used Figure 44 to see what happened with various inputs. If I had just leaped into trying to implement the Get_Command_Line procedure, and found that Show didn't work, I wouldn't have been able to tell if the error was in Show or Get_Command_Line. The two test stubs were so simple, I could have more confidence that they were correct, and could focus my attention on Show, where the problem probably was.

Test stubs don't need be limited to fixed data or user- supplied data. You can write test stubs that take input data from a file. Test stubs need not be just input simulators. They can also display or record data sent to them. Sometimes stubs count how many times they are called, or record the maximum or minimum values they receive. They can set a flag after they have been called a certain number of times. What you can do with a stub is limited only by your imagination.

5.4.3. Using Stubs and Drivers

Let's use and example to show how drivers and stubs can be used to check a program. Suppose you are

Figure 44. Get_Command_Line stub 2.

```
-------------------------------------------------------
--             GCLS2.ada
--             9 June 1987

--             Do-While Jones
--             324 Traci Lane
--             Ridgecrest, CA 93555
--             (619) 375-4607

-- Get_Command_Line, Stub 2


with TEXT_IO;
procedure Get_Command_Line(TAIL   : out string;
                           LENGTH : out natural) is
  TEXT : string(1..80);
  L    : natural;
begin
  TEXT_IO.put("What's on the command line? ");
  TEXT_IO.get_line(TEXT,L);
  TAIL(1..L) := TEXT(1..L);
  LENGTH := L;
end Get_Command_Line;
```

Figure 45. Lookup Driver.

```
------------------------------------------------------------
with SCROLL_TERMINAL; use SCROLL_TERMINAL;
with Lookup;
procedure Lookup_Driver is
  FLIGHT_NUMBER, REEL : integer;
  FLIGHT : string(1..5);
begin
  put_line("Lookup Driver");
  loop
    new_line;
    get("What flight? (0 to quit) ",FLIGHT);
    FLIGHT_NUMBER := integer'VALUE(FLIGHT);
    exit when FLIGHT_NUMBER = 0;
    Lookup(FLIGHT_NUMBER, REEL);
    put_line("Flight"
      & integer'IMAGE(FLIGHT_NUMBER)
      & " is on reel" & integer'IMAGE(REEL));
  end loop;
  put_line("Done.");
exception
  when PANIC =>
    put_line("Done.");
end Lookup_Driver;
```

writing a program that replays data from a test flight. The user enters the number of the flight he wants to replay. A procedure then searches a data base to see which reel of tape contains the data for that flight, and tells the computer operator to mount that reel. Suppose the wrong reel is consistently mounted. Where is the problem?

If you think the problem is in the routine that looks up the reel number for a given flight, you can use a test driver. The test driver could be as simple as the one shown in Figure 45. It asks you for a flight number and tells you what reel it is on. It will tell you if the Lookup routine works or not.

If there are symptoms that suggest that the Lookup routine is working correctly, is getting wrong input data, or maybe isn't being called at all, you can replace it with a stub like the one shown in Figure 46. Whenever it is called, it writes, "What reel contains flight number XXX? ", where XXX is the parameter that was passed to the stub. You enter YYY and press return. Then you can isolate the problem. Was XXX the flight number the user entered, or was the wrong parameter passed to the stub? Did the operator get a message to mount reel YYY, or was he told to mount a different reel?

5.4.4. Test and Demo Programs

The disks containing the source code for the listings also include some test and demo programs. There is a subtle distinction between the two. A test program is thorough, and usually doesn't involve much operator intervention. The Coordinates_Test program in Figure 42 is called a test program because it thoroughly tests the COORDINATES package and prints the results.

A demo program is just a quick confidence check. It may not check every part of the package, and it

Figure 46. Lookup Stub.

```
------------------------------------------------------------
with SCROLL_TERMINAL; use SCROLL_TERMINAL;
procedure Lookup(FLIGHT : integer;
                 REEL   : out integer) is
  REEL_NAME : string(1..6);
begin
  put("What reel contains flight");
  put(integer'IMAGE(FLIGHT));
  get(" ? ", REEL_NAME);
  REEL := integer'VALUE(REEL_NAME);
end Lookup;
```

usually involves a user interface. For example, a demo program might ask the user to enter an angle in degrees, and then print the sine, cosine, and tangent of the angle. The user can run this program a couple of times and observe the results, just to see if the program compiles and runs.

5.5. The Cost of Testing

It is well known that most of an iceberg is hidden under water. The same is true of software. The size of a software project is often described by the number of lines of code in the product, but that's just the tip of the iceberg. The amount of software that needs to be written to test the product can be staggering. Often the number of lines of code of test software will exceed the lines of code in the product software.

Test software can get out of hand in a hurry. Suppose you have to write three lines of test software for every new line of software you deliver. (That's probably what I average.) Then suppose management says that all your test software must be fully tested. (You've got to know your test suite works, don't you?) So for every line of test software you write, you need to write 3 lines of software that tests the software that tests your product. If X is the number of lines in your product, you will have to write 3X lines of test software, and 9X lines of code that tests the test software. Your job has just increased by a factor of 12! If management then insists you fully test the software that tests the software that tests your product, your job (and cost and schedule) has increased by a factor of 39. Sooner or later you have to call an end to the madness. (Fortunately, the managers I've worked for have required formally testing the test software, and no more.)

What generally happens is that people write as much test software as they can in the time left over at the end of the project. (Is there ever time left over at the end of the project?) The test software is inadequate, so the product gets shipped with some bugs in it.

Maybe you don't believe you have to write more test software than product software. Well, just look at size of the Cos_Test (Listing 74) and Cos_Diff (Listing 75) compared to the fraction of the TRIG package they are testing. I wanted to include complete test programs for all the source code in this book, but there's just not enough room for it all. That's why I've only described Cos_Test.

Testing is a big job. You have to plan money, people, and most of all, TIME for it. If you write three lines of test code for every line of product code, it will take you three times as long to write the test code. Don't expect to write and debug all your test code in the last month before the critical design review. Even if you could, it wouldn't do you much good. By that time the design is cast in concrete (and probably behind schedule), so nobody is going to change anything unless your test programs find catastrophic errors. If you find moderate errors, people will say, "That's a shame, but we can't do anything about it now. Why didn't you catch these errors sooner?" You have to code and test early in the development cycle while there is still time to take corrective action. Some people believe it is possible to do such good planning during the design phase that flawless code can be written in a few weeks at the end of the project. They think testing is a mere formality to show that the design is correct. That's nonsense.

Chapter 6. CONCLUSION

I've tried to give you the benefit of years of experience in a few pages. You can use it, ignore it, or build on it. It's up to you.

If you only learn one lesson from this whole book, I hope it is this: You can make your job much easier by filling you bag-of-tricks with reusable software components. Then most of your work reduces to simply putting those building blocks together to make whatever you want. It becomes child's play, like building something out of Tinker Toys.

If you do this, your job becomes more fun because you eliminate a lot of the drudgery. You don't keep solving the same old problems over an over. You use solutions you've already found for those problems, and devote most of your time to solving newer, more challenging problems. You cut down on the time you spend testing and documenting your software because many of the components have been tested and documented already.

The only way you can make this work is by learning to write independent modules. You have to hide special operational details inside a black box where they can't be seen. Start with modules that are small and simple, then build the smaller modules into bigger ones. Control the flow of information between modules by using parameter lists whenever possible.

This method works. I've used it in FORTRAN, assembly, and HPL. It works especially well in Ada, because Ada was designed to support this way of writing software. Let it work for you.

Chapter 7. EPILOG

I finished writing Ada in Action in February, 1989. Now it is January, 1995, almost six years later. If I were to write Ada in Action all over again, what would I write differently? Not much. That's why this second edition is nearly identical to the first. But I have learned a little in the last few years. There are a few things I could have done better.

7.1. Standard Size STANDARD_INTEGERS

I have added representation clauses to the type definitions in STANDARD_INTEGERS. There were times when I needed to interface with operating system data structures, and I needed to be sure that Integer_16 really was just sixteen bits.

7.2. New Naming Conventions

I have abandoned the file naming convention I used to use. The old convention was based on initials. For example, the TRIG package specification was in file TS.ada, and the TRIG package body was in file TB.ada. Some of the file names (like ftbcgfp.ada for FORM_TERMINAL.Create. Get_Field.Protect_Field) were bizarre to say the least.

The last straw came when I wrote a magnetic tape interface package interface called TAPE. Naturally, I called its specification TS.ada, and the body was in TB.ada. I happened to copy these files into a directory that contained the TRIG files, and clobbered them.

I now use a file naming convention based on part numbers that I assign to reusable software components. For a complete description of the new convention, see the file pub/users/do_while/components/names.txt.

I have also abandoned the type naming convention (described in section 2.10) which uses abbreviations for real types and full names for integer types. It was more trouble than it was worth.

7.3. Software Documentation

I've adopted a more structured format for the software documentation. For each software component, I write a four- part document.

Part 1 is the Programmer's Guide. It tells what the software component does, and how to use it. It is really an expanded description of the component's specification.

Part 2 contains Implementation Notes. It tells how the software component works, with special emphasis on why design decisions were made. If other approaches were considered and rejected, they are described and an explanation is given why they weren't used. (Sometimes the explanation is, "They all seemed to be pretty much the same, but I had to pick one, so I did.")

Part 3 is titled Suggested Improvements. When writing the first two parts, I often think of things I could have done better. I document them in Part 3, and usually implement them the next time I revise the component. Eventually, Part 3 just contains the word, "None."

Part 4 takes two different forms, depending on the situation. The hard-copy document contains complete source listings and all associated data files for the component. When I put the document on the Internet, however, I change Part 4 to the list of files that should be in Part 4. All the files referenced in Part 4 are in the same FTP directory, so you can create a hard-copy document by catenating the listed files to the end of the documentation file. (That's easier than un-catenating the files from the end of the document so you can compile them.)

My software components are grouped in families. There is always a document for part number "00" of the family, which is really an overview of all the components in the family. It briefly lists all the components in the family and tells, in a sentence or two, what they do. It ends with a suggested compilation order.

7.4. Separate Non-abstract Data Types from Subprograms

It is customary to encapsulate data types with their subprograms in a package. This is certainly a good thing to do for abstract (private and limited private) types. It creates a clean, complete abstraction that is easy to maintain.

But what is good for abstract types isn't necessarily good for other types. Consider the TRIG package. I defined the data types Deg and Rad in the TRIG package. I later discovered that there were many times when I wanted to declare objects of type Deg or Rad when I had no intention of doing any trigonometry at all. I just wanted to add some angles together. I had to "WITH TRIG" and "USE TRIG" to make the addition operation visible. This had the side effect of making all the TRIG subprograms visible, and possibly added a lot of "dead code" to the program.

I ran into a similar situation when I wrote a package called BASIC_GRAPHICS which defined the data types X_coordinates and Y_coordinates that defined locations on the screen. I often found myself USEing BASIC_GRAPHICS just to make the type definitions of the coordinates visible. It really seemed strange to me that the MENU package should have to be compiled in the context of a BASIC_GRAPHICS package that draws circles and polygons, and many other things that have nothing to do with displaying a menu (other than the fact that you have to tell the menu where to appear on the screen, which requires an x/y coordinate point).

I now separate visible type definitions from operations. That is, the ANGULAR_UNITS package (Physical Units part number PU08) defines the types Degrees and Radians. The new specification of TRIG (Ada in Action part number AA09) is compiled WITH ANGULAR_UNITS so it can USE Degrees and Radians. Other packages can be compiled WITH ANGULAR_UNITS to get just the data types and primitive operations, without unnecessary TRIG functions. Similarly, X_coordinates and Y_coordinates are now defined in BASIC_GRAPHICS_TYPES (Graphical Interfaces part GI03) rather than BASIC_GRAPHICS (part GI04).

7.5. ASCII_UTILITIES is too Broad

I put too much unrelated stuff in ASCII_UTILITIES. If I had it to do over again, I would have made Fixed_Image, Float_Image, and Value three separate library procedures. (I would have called the Value function Real_Value instead.) I would have put Number_bases, Value, and Image in a package called DIGIT_CONVERSIONS. (If I had done that, there would have been no need for the discussion of qualified expressions in section 3.1.2.)

There isn't really any harm in putting all this unrelated stuff in ASCII_UTILITIES, but it doesn't set a very good example for readers.

7.6. Finalization

I first ran into a need for finalization when I ported the VIRTUAL_TERMINAL to the Encore computer running the MPX- 32 operating system. I ran into it again when I ported the VIRTUAL_TERMINAL to UNIX. In both cases it was necessary to change some console interface parameters.

For example, in UNIX, I had to turn off the character echo and disable the canonical mode so I could get unfiltered characters from the keyboard as soon as keys were pressed. This was easily done by putting the appropriate instructions at the end of the VIRTUAL_TERMINAL package body, where it would be executed automatically at elaboration. The problem was that I had to remember to reset the console interface parameters when the program finished, or else the UNIX command line interface would not work!

I encouraged the Ada 9X team to include package finalization in Ada 95. If they had taken my suggestion, I could have written the VIRTUAL_TERMINAL package body this way.

```
package body VIRTUAL_TERMINAL is
     [lots of stuff deleted]
begin
     Echo_Off;
     Canonical_Off;
at end
     Echo_On;
     Canonical_On;
end VIRTUAL_TERMINAL;
```

The procedures Echo_Off and Canonical_Off would be called automatically during the elaboration of VIRTUAL_TERMINAL. When the package passed out of scope, package finalization would automatically call Echo_On and Canonical_On. The application programmer would not have to worry about them.

I lost that battle because package finalization is inadequate for some applications. Specifically, there are cases when finalization needs to be done on an object basis rather than a package basis. When an object goes out of scope, it needs to be finalized even though the package that defines the type is still in scope. Package finalization can be done by declaring a dummy object in the package body and associating finalization procedures with the dummy object. When the package goes out of scope, then the object goes out of scope, and finalization gets done. I don't think that's a very elegant solution, but I must admit that it is acceptable.

But the VIRTUAL_TERMINAL finalization problem needed an immediate solution. Two years ago I could not wait for whatever finalization process Ada 9X would eventually have (if any). I had to solve the problem with Ada 83, so I did.

Eventually, I concluded that any kind of implicit finalization could get me into trouble, so it is probably best to avoid it completely.

When I was developing some air-traffic control software, I discovered that multiple objects needed shared access to devices such as the keyboard, screen, and mouse. When a menu goes away, it doesn't need the mouse any more, so it tries to close the mouse. But there might be a control panel or a dialog box that still needs the mouse, so the mouse should stay open even though the menu tried to close it. Implicit finalization might close something that should remain open.

I finally decided that the approach that gives me the most flexibility is to have explicit Open and Close procedures for all shared resources. The package body contains a hidden reference counter that is initially set to 0. If Open is called when the reference counter is 0, it opens the device and sets the reference counter to 1. If Open is called when the reference counter is not 0, it simply increments the reference counter without opening the device. If Close is called when the reference counter is 1, then it closes the device and sets the reference counter to 0. If Close is called when the reference counter is greater than 1, it just decrements the reference counter without closing the device.

All components that use shared devices must follow the "kindergarten convention." That is, "If you opened it, close it!"

This approach gives me three options. I can let the main program open the keyboard, mouse, and screen at the beginning of the program, and then close them at the end; or I can let every subprogram that needs a resource open it when it needs it and close it when it is done with it; or I can do both. (I generally select the third option.)

7.7. Ada 95

Now that Ada 9X has become Ada 95, I've been asked if I will re-write Ada in Action with an emphasis on the new Ada 95 features. No, I won't in the foreseeable future.

I write about what I have learned from personal experience. I don't repeat fables I've read someplace else. There are no validated Ada 95 compilers yet, so I don't have any experience with them. GNAT, an evolving Ada 95 compiler has been out for a year or so, but I don't enjoy working on quicksand, so I haven't used it. I'm not even anxiously awaiting the arrival of a validated Ada 95 compiler because I don't have an immediate need for any of the new features. Eventually I will probably come across a real application where one of the new Ada 95 features will permit me to write a more elegant solution than Ada 83 did, but I haven't found one yet.

For example, in section 2.4.2 I said, "Although Ada always allows you to add new operations to a data type, she never lets you take away operations derived from a parent." That is no longer true. Ada 95 lets you make operations abstract, which effectively takes them away. So I could have created the dimensional units types by starting with integer or float types and taken away multiplication and division. But which is better? to have a specification that lists all the operations a type has; or to have a specification that lists all the operations a type doesn't have? I'd prefer to know what it can do, not what it can't. So I'm not about to go back and rewrite the dimensional data types and use abstract operations.

The most important thing I learned from following the progress of the Ada 9X project is that Ada had very little room for improvement. There were hundreds of suggested improvements to Ada, but the vast majority of them were rejected because Ada already did the right thing.

Most of the changes that were made were simply syntactic sugar. There were, in my opinion, only five significant changes.

The first significant change was the expansion of the character type from 7-bits to 8-bits, permitting the use of accented characters. There no doubt is a need for this given the popularity of Ada outside the United States.

The second change (really a group of changes) provided for more programmer control of real-time performance. Although nearly all of my work for the past 18 years relates to real-time data processing and display (with cycle times much less than 100 milliseconds), I have never experienced the problems that these changes were supposed to solve. Therefore, I don't feel qualified to comment on whether these changes have merit or not.

I campaigned vigorously against the third change, the hierarchical package structure. Although I agree that it is good to break huge, monolithic packages into smaller pieces, I fear the "child package" solution invites abuse.

Years ago I called the potential problem the "Howard Hughes effect." This was shortly after wealthy Mr. Hughes died without leaving a will, and several people claimed to be his children and therefore rightful heirs to his fortune. I tried, without success, to make Ada 95 require the parent package to contain a list of all legitimate children, thus limiting the child packages that could be written. I still believe this is necessary to avoid an abuse that results in loss of privacy. But the prevailing opinion was that specifying the names of children would make Ada 95 more restrictive than C++. That restriction would be perceived as a weakness which would become a marketing disadvantage.

If you write a package that declares a private type in the package specification, or a type hidden in the body, Ada 83 prevents me from writing any code that takes advantage of representation details (unless I instantiate the conspicuously evil UNCHECKED_CONVERSION function). This gives you the freedom to change the representation without any possibility of messing up my code. That's what privacy is all about.

Ada 95, however, allows me to claim that my package is a child of yours, which lets me see your private parts (whether you want me to or not). My child package can take advantage of your representation details without using UNCHECKED_CONVERSION. You can declare things to be private, but it means nothing. I can now invade your privacy at will.

The argument against my objection is that if the private type changes and the parent is recompiled, then the child packages become obsolete and have to be recompiled. During that recompilation Ada will catch any inconsistencies.

My counter argument is that I can (and do) achieve the exact same effect in Ada 83. I declare private things that I intend to use in multiple packages in a package with a name that begins with "PRIVATE_".

For example, I have a family of network interface components that includes packages called NETWORK_CLIENT, NETWORK_SERVER, NETWORK_TRANSMITTER, and NETWORK_RECEIVER. (The CLIENT/SERVER pair does point-to-point communications, the TRANSMITTER/RECEIVER pair does one-to-many broadcasts.) The package specifications present the application programmer with simple, virtual abstractions of interfaces to other parts of the program running on other computers. The package bodies hide the actual interface, which may be shared memory, remote procedure calls, serial RS-232, parallel DR11-W, GPIB, ethernet connections, or something else.

The package bodies I commonly use on the Sun and SGI computers depend on interfaces to UNIX services (connect, bind, accept, send_to, etc.). I declare the implementation- specific ethernet interfaces in a package called PRIVATE_NETWORK_DETAILS. The specification of this package and the accompanying documentation loudly proclaim that no application program should ever be compiled WITH PRIVATE_NETWORK_DETAILS. My development standards treat the PRIVATE_NETWORK_DETAILS package like UNCHECKED_CONVERSION and UNCHECKED_DEALLOCATION. Using any of them is considered dangerous, and requires special dispensation.

But suppose some evil person does use the forbidden PRIVATE_NETWORK_DETAILS, and I make changes to it, then Ada 83 will make those illegitimate dependents obsolete as soon as I recompile PRIVATE_NETWORK_DETAILS. The effect is exactly the same as in Ada 95 if someone falsely claims to be a child. So, in my opinion, Ada 95 gives me nothing that I didn't already have in Ada 83, and takes away my right to privacy.

I have mixed feelings about the fourth change, which added C++ style object-oriented features. People seem to be having great difficulty coming up with practical examples of how the features can be used. This suggests to me that they solve interesting academic problems, but do not have much practical utility. I used to think that these features were necessary for dynamically created user interface windows. But for the last two years I have been writing an air- traffic control program that lets the user view the data using any combination of twelve different display windows, which can be created and modified in real time. It has been very easy to do in Ada 83, and I don't think Ada 95 would make it any easier.

I do think the object-oriented features are important to Ada because of the "rhythm rail effect." In the 1960's, I used to teach guitar at a music store that sold Hammond organs. The organs sold in the other music store in town featured glorified metronomes called rhythm rails. (They were the fore-runners of drum machines.) Hammond didn't offer any organs with rhythm rails. Customers would buy inferior organs from the other store because the rhythm rail seemed like such a neat gadget. After a few weeks, they never used the rhythm rail. It was too hard to play along with it. But by then, they had already bought the organ. Hammond eventually added rhythm rails to their organs, because they found they could not compete with the junk organs unless they had them, too.

Object-oriented features are today's rhythm rails. Nobody will buy a language without them. Lots of people are playing with Ada 95 prototypes just to get experience with the new object-oriented features. The novelty will wear off soon, but before it does, people will have learned about Ada's other, more useful features. After playing with Ada 95 for a while, they won't be afraid of her any more, and will appreciate her many other virtues. So, the new object- oriented features are important because they have made Ada more interesting and novel, and are getting potential customers into the store. Many people will adopt Ada because of them, but I don't expect them to be used very much in actual programs.

The fifth important change introduced in Ada 95 doesn't get much publicity, but it may be the most important. Ada 95 allows subunits to have the same simple name, as long as the expanded name is unique. I think this will be very useful in big projects, where it is likely that several subunits written by different people will have common names (like Get and Put). You won't be frustrated because you can't call the subunit what you want to call it because someone else has already used that name in another part of the program that isn't even visible. It is a feature you will use without knowing that you are using it, unless you later try to compile your program on an Ada 83 compiler, but it will make programming much easier.

7.8. Estimating Completion

I'm still looking for better ways to tell if I am on schedule, and how much time projects will take, than the method described in section 4.6.22.3. I have a novel approach, and I'm still collecting data, but I think I'm on the right track, so I'll tell you about it. It is becoming an integral part of my software development process.

After the usual thrashing around, the requirements specification gets signed off. From the requirements specification I write the user's guide and one or more interface specifications. (The number of interface specifications depends on the number of external devices the program interacts with.)

Generally the users are surprised and disappointed with the user's guide. I show them how it does exactly what they approved in the requirements specification. They say that they didn't know that was what the precise legal language in the requirement specification meant. So, I rewrite the user's guide to satisfy them. (A similar process usually happens with the interface specifications, too. For convenience I won't explicitly refer to the interface specifications any more; but whenever I talk about the user's guide, understand that I mean the user's guide and interface specifications.)

From this point on, the user's guide drives the design. It is the de facto requirements document. I do not waste time revising the requirements specification. The requirements specification's only purpose was to lead to the initial user's guide. The requirements specification has served its purpose and is shelved. (Fortunately I'm not in a situation where the requirements specification is a binding legal document describing the delivered software.)

I change all the characters in the user's guide to italics. Text written in italics describes things that the prototype does not do yet. As the prototype evolves, I change the text describing the implemented features back to normal font. When the program is complete, the entire user's guide will be in normal font.

I store a copy of the user's guide in Microsoft's Rich Text Format (RTF). I then run a simple program that scans the file, counting the total number of characters and the number of characters that are italic. It computes the percentage complete by dividing the number of characters that are not italic by the total number of characters in the document.

I code and test the prototype for three months. At the end of the three months (regardless of the progress that has been made), I freeze the design and fully document the prototype. I revise the user's guide based on experience with the prototype, and change the sections of the user's guide that have been implemented in the prototype from italic to normal characters. I store the revised user's guide in RTF format and let my program count the characters and compute percentage of completion.

In general, the number of non-italic characters increases with each revision. The number of total characters also increases because experience with the prototype leads to new requirements.

I do not yet have enough data to know what my normal rate of completion is. I don't expect it to be linear (and it hasn't been). It will be years before I have completed enough projects to compute meaningful statistics. If you work in a large company where dozens of projects are going on at once, you may be able to take similar results and come up with results sooner.

I have noticed some interesting correlations in some of the measurements I have taken so far. Scatter diagrams of number of bytes of source code as a function of number of lines of source code fall very close to a line with a slope of 30 to 1. The number of bytes of executable code per line of source code is generally 60 to 1. That is, 38,000 lines of source code generally results in roughly 1.14 Mbytes of source code and 2.28 Mbytes of executable code, with optimization turned off. These numbers are easy to measure. Do you get similar results?

I'm also looking at bytes of documentation per byte of executable code, number of bytes of source code per non- italic character in the user's guide, and so on. I'm measuring hours spent and dollars spent, so I can compute lines of code per hour and dollars per line of code (or dollars per byte of executable code).

Some ratios change depending on the phase of the project. For example, at the beginning of the project progress is measured in lines of code produced per day. Near the end of the project, when redundant routines are being consolidated in subroutines or generics, and more efficient algorithms are found, progress should be measured in lines of code REMOVED per day. Near the end of a project, a decrease in the ratio of lines of code per day is good.

It will probably be several more years before I have enough data to know what it all means. It is not a good idea to make any vast changes based on half-vast data. So, for now, I'm just collecting the data. I suggest you do the same, and see if you discover anything interesting.

The most important thing I've learned about predicting software cost and schedule is this: You can't predict how much your next project will cost, and how long it will take, unless you can tell how much your last project cost, and how long it took. If it cost $150 per line of code on your last project, it will probably cost $150 per line of code on your next project. If you produced an average of 3 lines of code per person per day on your last project, you will probably produce 3 lines of code per person on your next project. So, take good data on your current project, and try to reconstruct data from your past projects, and you will have a better idea of how to predict progress on your next project.

7.9. Predictions

John Wiley & Sons did very little editing on the first edition, but there were two paragraphs dealing with design methodologies that they strongly urged me to take out. They felt I was overly critical of MIL-STD-2167A, and that my rejection of a standard worthy of sainthood damaged my credibility. All the experts said MIL-STD-2167A was the one true way to develop software, so I would look like a fool to criticize it.

I was just ahead of my time. Now 2167 has largely been repudiated. I wish that I had fought to keep these two paragraphs in the first edition.

> One of the real problems with the "waterfall model" of the software life cycle described in MIL-STD-2167A, is that it puts coding off until the last moment before delivery. It naively assumes that test routines have been written from logical design and physical design products, and are ready and waiting to test coded modules as they come off the assembly line. It just doesn't work that way. By the time the modules finally get coded (often weeks or months behind schedule), they no longer match the design products. The testing people are ready to test modules that were never written, and don't have anything that will test what was written.
>
> I don't recommend the MIL-STD-2167A approach for software development in any language, and especially not for Ada. Other languages may make it hard to code modules until the end of the development phase, but Ada doesn't. Ada lets you prototype early. You can write routines that test the prototype and let them evolve into the final test suite just as the prototype evolves into the product.

Now that I don't have an editor to restrain me, I can make the following predictions.

C and C++ have established such a presence that they will be around as long as there are computers running UNIX. There will always be significant software development in C++, but I don't think C++ will retain its current popularity. It is based on C and a questionable object-oriented model, and that will be its downfall. Maintenance of large C++ programs will be very difficult. C++ has been around long enough that we are starting to hear the first of what will be a long series of spectacular failures. I expect the computer magazines to publish more articles critical of C++ in the next few years, and then C++ will take its place along side FORTRAN and COBOL as a heavily used, but seldom discussed, language.

I REALLY LIKE the Software Engineering Institute's software Capability Maturity Model. There is a lot of good advice in it. An organization that is following the CMM processes and is using Ada will be a strong competitor.

7.10. Final Thoughts

On June 6, 1982, I took a three-day course called "Developing Software with Ada" taught by George Cherry. The government had previously tried to force me to use a horrible language called ATLAS, and it appeared that they were about to try to make me use Ada. I took the Ada course to find out all of Ada's weaknesses so I could convince my managers not to make me use Ada. By the second day of the class, I was in love with Ada.

As it turned out, I had trouble getting the government to let me use Ada. It was 1989 before I could even get my working group to buy a compiler, and I wasn't permitted to officially write Ada code until 1993. (Before that, I wrote and tested algorithms in Ada, and translated them to FORTRAN.)

But I made a personal commitment to Ada in 1982, and cut my teeth on a remarkable unvalidated Ada compiler called Maranatha A (which ran on a CP/M computer with 56 Kbytes of memory) written by David Norris. I later bought an IBM PC AT clone and got several PC Ada compilers. I expected to ride Ada's popularity all the way to the top, as the whole world embraced Ada.

Well, Ada didn't achieve the success she deserved. Some of the first compilers were pretty bad, and they gave her a bad reputation. The DoD Mandate was largely ignored, and only inspired hate and resistance. (Programmers said, "If Ada were any good, the government wouldn't be forcing it down our throats.") The acceptance of Ada has been much less than I expected.

Despite my disappointment in Ada's market share, I don't regret choosing Ada. Certainly "C in Action" would have sold far more copies than Ada in Action did, but my goal wasn't to make the big bucks. I guess I'm just an artist at heart. My goal is to be the best, not the richest.

Ada has helped me be the best that I can be. I've been far more productive using Ada than I could have if I used any other language. My Ada code is easier to understand, more reliable, and easier to maintain than my FORTRAN or assembly language code is. But beyond that, Ada has helped me to help others be the best they can be. That's what is most important to me.

So, I will continue to use Ada.